

CLARION 5

**Programmer's
Guide**

**COPYRIGHT 1994, 1995, 1996, 1997, 1998 by TopSpeed Corporation
All rights reserved.**

This publication is protected by copyright and all rights are reserved by TopSpeed Corporation. It may not, in whole or part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form without prior consent, in writing, from TopSpeed Corporation.

This publication supports Clarion 5. It is possible that it may contain technical or typographical errors. TopSpeed Corporation provides this publication “as is,” without warranty of any kind, either expressed or implied.

TopSpeed Corporation
150 East Sample Road
Pompano Beach, Florida 33064
(954) 785-4555

Trademark Acknowledgements:

TopSpeed® is a registered trademark of TopSpeed Corporation.

Clarion for Windows™ is a trademark of TopSpeed Corporation.

Btrieve® is a registered trademark of Pervasive Software.

Microsoft® Windows® and Visual Basic® are registered trademarks of Microsoft Corporation.

All other products and company names are trademarks of their respective owners.

CONTENTS SUMMARY

FOREWORD	29
PART I—CLARION LANGUAGE PROGRAMMING	31
PART II—ADVANCED PROGRAMMING RESOURCES	117
PART III—TEMPLATE LANGUAGE PROGRAMMING	355
PART IV—ISAM DATABASE DRIVERS	551
PART V—SQL ACCELERATORS	685
INDEX	835

CONTENTS

FOREWORD	29
PART I—CLARION LANGUAGE PROGRAMMING	31
1 - PROGRAM STRUCTURE	33
Structured Programming	33
PROCEDURES	33
Local Stack-Based Data Declarations	33
PROGRAM MAP	34
MODULE	34
MEMBER	35
MEMBER MAPs	36
MODULEs within MEMBER MAPs	37
PROCEDURE MAPs	38
Summary	39
2 - EASING INTO OOP	41
Opening Shots	41
Procedural Code Re-Visited	41
CLASS Declaration	43
Encapsulation	43
Inheritance	48
Polymorphism	54
Local Derived Methods	57
Wrap Up	58
3 - OBJECT ORIENTED PROGRAMMING (OOP)	59
Object Overview	59
What are Objects?	59
Why Objects?	59
What Makes an Object?	60
Clarion's OOP Extensions	62
The CLASS Structure—Encapsulation	62
Derived CLASSes—Inheritance	68
Virtual Methods—Polymorphism	73
Summary	75

4 - DATABASE DESIGN	77
Database Design	77
Relational Database Design	77
File Relationships	79
Translating the Theory to Clarion	80
Referential Integrity	82
Summary	86
5 - DATA FILE PROCESSING	87
Data File Processing	87
File Access Methods	87
KEY and INDEX	87
Sequential File Access	88
Random File Access	91
Summary	92
6 - MULTI-USER CONSIDERATIONS	93
Multi-User Considerations	93
Opening Files	93
Concurrency Checking	94
HOLD and RELEASE	99
LOCK and UNLOCK	101
"Deadly Embrace"	103
Summary	104
7 - DEVELOPING CLIENT/SERVER APPLICATIONS	105
Introduction to Client/Server	105
Client/Server Defined	105
Types of Client/Server Database Applications	105
SQL Database Engines	106
Clarion and SQL	106
Database Design and Network Traffic	108
Referential Integrity Handling	108
Data Validation	109
Clarion Language Client/Server Support	110
The VIEW Structure	110
The BUFFER Statement	111
Embedded SQL in Clarion	112

NULL Data Handling	115
Error Handling	116

PART II—ADVANCED PROGRAMMING RESOURCES..... 117

8 - CUSTOMIZING THE DEVELOPMENT ENVIRONMENT 119

An Overview 119

Command Line Parameters	119
-------------------------------	-----

Non-Modifiable Clarion4.INI File Sections 120

User Information	120
Paths	120
Environment Options	120
The Clarion Applets	122

Modifying the Clarion Environment 123

Specifying User Defined Applications	123
Adding choices to the Clarion Menu	124
Adding choices to the Clarion Setup Menu	125
Adding File Masks to File Types Drop Down lists	126
Adding Tabs to New, Open, or Pick File Dialogs	127
Specifying Make File Types	129
Print Specifications	129

Environment Option Settings 130

Auto Populate Options	130
Dictionary Options	131
Project System Options	133
Application Generator Options	134
Dictionary Synchronization Options	137
Template Registry Options	140
Window Formatter Options	141
Control Default Size Options	143
Report Formatter Options	145
Editor Options	147
Editor Tabs	148

9 - USING CLARION AS A DDE SERVER 149

Overview 149

Connect to Clarion as DDE Server	151
Disconnect from the Clarion DDE Server	152

Export a Dictionary to Text (TXD) format	153
Import a Dictionary from Text (TXD) format	154
Export an Application to Text (TXA) format	155
Import an Application from Text (TXA) format	156
Load an Application	157
Generate an Application	158
Execute a Project or Application	159
Running a Utility Template	160
Registering a Template Class	161
Getting DDE Error Messages	162
Clarion DDE Errors	163
DDE Service Errors and associated messages	163

10 - .TXD FILE FORMAT 165

.TXD Files: Clarion Dictionaries in ASCII Format 165

.TXD File Organization 166

.TXD Skeleton	166
---------------------	-----

.TXD File Sections 168

[DICTIONARY]	168
[FILES]	170
[ALIASES]	184
[RELATIONS]	185

Common Subsections 190

[LONGDESC]	190
[USEROPTION]	190
[TOOLOPTION]	190
[SCREENCONTROLS]	191
[REPORTCONTROLS]	191

11 - .TXA FILE FORMAT 193

.TXA Files: Clarion Applications in ASCII Format 193

.TXA File Organization 194

.TXA Skeleton	194
---------------------	-----

.TXA File Sections 196

[APPLICATION]	196
[PROJECT]	198
[PROGRAM]—[END]	199
[MODULE]—[END]	200
[PROCEDURE]	201

Common Subsections	204
[COMMON]	204
[DATA]	205
[FILES]	209
[PROMPTS]	212
[EMBED]—[END]	216
[ADDITION]	219
12 - TOPSPEED'S PROJECT SYSTEM	223
Introduction	223
Project System Macros	225
Setting Macro Values	226
Special Project System Macros	227
Basic Compiling and Linking	230
Specifying Compile and Link Options	230
Compile and Link Commands	232
Conditional Processing and Flow Control	236
User Interface	237
Boolean Expressions	237
The Redirection File	239
File Management	241
Other Commands	243
TopSpeed #pragmas	245
Pragma Syntax	245
Pragma Classes	246
Call #pragmas	247
Data #pragmas	252
Check #pragmas	256
Name #pragmas	258
Optimize #pragmas	259
Debug #pragmas	262
Module #pragmas	263
Option #pragmas	265
Warn #pragmas	266
Project #pragmas	273
Link #pragmas	274
Link_Option #pragmas	274
Define #pragmas	276
Pre-defined Flags	278

Project System Examples	279
Module Definition Files (.EXP Files)	287
16-Bit	287
32-Bit	297
13 - MULTI LANGUAGE PROGRAMMING	303
Overview	303
Compiler Integration	304
Integrating 3GL Modules into Clarion Projects	304
Resolving Data Types	306
C and C++ Data Type Equivalents	307
Modula-2 Data Type Equivalents	310
Pascal Data Type Equivalents	312
Prototyping 3GL Functions in Clarion	314
Parameter Data Types	315
Return Data Types	316
Passing Parameters	317
Resolving Calling Conventions	318
Resolving Naming Conventions	319
The EXTERNAL and DLL Attributes	321
Programming Considerations	322
Using C++ Class Libraries	322
Summary	323
14 - API CALLS AND ADVANCED RESOURCES	327
Prototypes and Declarations	327
Clarion to C/C++ Standard Library	327
Clarion to Windows API	327
Modula-2 to Clarion	328
C/C++ to Clarion	328
Accessing Clarion's Runtime Library from C/C++ or Modula-2 Code	329
Structures and Data Type Definitions	329
Run-Time Variables	329
Clarion Built-in Procedures	330
Standard C Functions in Clarion's Runtime Library	343
Conversion Functions	343
Integer Math	344

Char Type Functions	344
Utility Functions	347
String Functions	347
Low-Level File Manipulation	351

PART III—TEMPLATE LANGUAGE PROGRAMMING 355

15 - INTRODUCTION TO TEMPLATE LANGUAGE 357

Template Language Overview	357
What Templates Are	357
Template Types	359
What Templates Do	359
Pre-Processing and Source Code Generation	360
Embed Points	361
Template Prompts	362
Data Dictionary Interface	363
Template Structure	364
Template Source Format	364
The Template Registry File	365
Customizing Default Templates	366
Adding New Template Sets	368

16 - TEMPLATE ORGANIZATION 369

Template Code Sections	369
#TEMPLATE (begin template set)	369
#SYSTEM (template registration and load)	370
#APPLICATION (source generation control section)	371
#PROGRAM (global area)	373
#MODULE (module area)	374
#PROCEDURE (begin a procedure template)	375
#GROUP (reusable statement group)	376
#UTILITY (utility execution section)	378
#CODE (define a code template)	379
#CONTROL (define a control template)	381
#EXTENSION (define an extension template)	385
Embed Points	388
#EMBED (define embedded source point)	388
#AT (insert code in an embed point)	391

#PRIORITY (set new embed priority level)	393
#ATSTART (template initialization code)	394
#ATEND (template reset code)	395
#CONTEXT (set template code generation context)	396
#EMPTYEMBED (generate empty embed point comments)	397
#POSTEMBED (generate ending embed point comments)	398
#PREEMBED (generate beginning embed point comments)	399
Template Code Section Constraints	400
#WHERE (define #CODE embed point availability)	400
#RESTRICT (define section use constraints)	401
#ACCEPT (section valid for use)	402
#REJECT (section invalid for use)	403
17 - DEFAULTS AND TEMPLATE DATA	405
Default Data and Code	405
#WINDOWS (default window structures)	405
#REPORTS (default report structures)	406
#LOCALDATA (default local data declarations)	407
#GLOBALDATA (default global data declarations)	407
#DEFAULT (default procedure starting point)	408
Symbol Management Statements	409
#DECLARE (declare a user-defined symbol)	409
#ALIAS (access a symbol from another instance)	411
#EQUATE (declare and assign value to a user-defined symbol)	412
#ADD (add to multi-valued symbol)	413
#DELETE (delete a multi-valued symbol instance)	414
#DELETEALL (delete multiple multi-valued symbol instances)	415
#PURGE (delete all single or multi-valued symbol instances)	416
#CLEAR (clear single-valued symbol)	416
#FREE (free a multi-valued symbol)	417
#FIX (fix a multi-value symbol)	418
#FIND (“super-fix” multi-value symbols)	419
#SELECT (fix a multi-value symbol)	420
#POP (delete and re-fix a multi-value symbol)	421
#SET (assign value to a user-defined symbol)	421
#UNFIX (unfix a multi-value symbol)	422

18 - PROGRAMMER INPUT**423****Input and Validation Statements****423**

#PROMPT (prompt for programmer input)	423
#VALIDATE (validate prompt input)	427
#ENABLE (enable/disable prompts)	428
#BUTTON (call another page of prompts)	429
#WITH (associate prompts with a symbol instance)	432
#FIELD (control prompts)	433
#PREPARE (setup prompt symbols)	434

#PROMPT Entry Types**435**

CHECK (check box)	435
COLOR (call Color dialog)	435
COMPONENT (list of KEY fields)	435
CONTROL (list of window fields)	436
DROP (droplist of items)	436
EMBED (enter embedded source)	437
EXPR (appended data fields)	437
FIELD (list of data fields)	438
FILE (list of files)	438
FORMAT (call listbox formatter)	439
FROM (list of symbol values)	439
KEY (list of keys)	440
KEYCODE (list of keycodes)	440
OPENDIALOG (call Open File dialog)	441
OPTFIELD (optional text or data field)	441
OPTION (display radio buttons)	442
PICTURE (call picture formatter)	442
PROCEDURE (add to logical procedure tree)	442
RADIO (one radio button)	443
SAVEDIALOG (call Save File dialog)	443
SPIN (spin box)	444
TEXT (text box)	444

Display and Formatting Statements**445**

#BOXED (prompt group box)	445
#DISPLAY (display-only prompt)	446
#IMAGE (display graphic)	446
#SHEET (declare a group of #TAB controls)	447
#TAB (declare a page of a #SHEET control)	448

19 - SOURCE GENERATION CONTROL	451
Template Logic Control Statements	451
#FOR (generate code multiple times)	451
#IF (conditionally generate code)	452
#LOOP (iteratively generate code)	454
#BREAK (break out of a loop)	456
#CYCLE (cycle to top of loop)	456
#CASE (conditional execution structure)	457
#INDENT (change indentation level)	458
#INSERT (insert code from a #GROUP)	459
#CALL (insert code from a #GROUP, without indentation)	460
#INVOKE (insert code from a named #GROUP)	461
#RETURN (return from #GROUP)	462
File Management Statements	463
#CREATE (create source file)	463
#OPEN (open source file)	464
#CLOSE (close source file)	465
#READ (read one line of a source file)	466
#REDIRECT (change source file)	467
#APPEND (add to source file)	468
#SECTION (define code section)	469
#REMOVE (delete a source file)	470
#REPLACE (conditionally replace source file)	471
#PRINT (print a source file)	472
Source Generation Statements	473
#GENERATE (generate source code section)	473
#ABORT (abort source generation)	473
#SUSPEND (begin conditional source)	474
#RELEASE (commit conditional source generation)	475
#RESUME (delimit conditional source)	476
#? (conditional source line)	477
#QUERY (conditionally generate source)	478
External Code Execution Statements	479
#RUN (execute program)	479
#SERVICE (TopSpeed internal use only)	479
#RUNDLL (execute DLL procedure)	480

20 - MISCELLANEOUS TEMPLATE STATEMENTS**483****Miscellaneous Statements****483**

#! (template code comments)	483
#< (aligned target language comments)	483
#ASSERT (evaluate assumption)	484
#CLASS (define a formula class)	484
#COMMENT (specify comment column)	485
#DEBUG (toggle debug generation)	485
#ERROR (display source generation error)	486
#EXPORT (export symbol to text)	487
#HELP (specify template help file)	487
#INCLUDE (include a template file)	488
#IMPORT (import from text script)	488
#MESSAGE (display source generation message)	489
#PROTOTYPE (procedure prototype)	490
#PROJECT (add file to project)	491

Built-in Template Procedures**492**

CALL (call a #GROUP as a function)	492
EXTRACT (return attribute)	493
EXISTS (return embed point existence)	494
FILEEXISTS (return file existence)	494
FULLNAME (return file path)	495
INLIST (return item exists in list)	495
INSTANCE (return current instance number)	496
INVOKE (call a named #GROUP as a function)	497
ITEMS (return multi-valued symbol instances)	498
LINKNAME (return mangled procedure name)	498
QUOTE (replace string special characters)	499
REGISTERED (return template registration)	499
REPLACE (replace attribute)	500
SEPARATOR (return attribute string delimiter position)	501
SLICE (return substring from string)	501
UNQUOTE (remove string special characters)	502
VAREXISTS (return symbol existence)	502

21 - TEMPLATE SYMBOLS**503****Symbol Overview****503**

Expansion Symbols	503
Symbol Hierarchy Overview	505

Built-in Symbols	506
Symbols Dependent on %Application	506
Symbols Dependent on %File	507
Symbols Dependent on %ViewFiles	509
Symbols Dependent on %Field	509
Symbols Dependent on %Key and %KeyOrder	512
Symbols Dependent on %Relation	513
Symbols Dependent on %Module	514
Symbols Dependent on %Procedure	515
Window Control Symbols	517
Report Control Symbols	519
Formula Symbols	520
File Schematic Symbols	521
File Driver Symbols	522
Miscellaneous Symbols	523
22 - ANNOTATED EXAMPLE TEMPLATES	525
Procedure Template: Window	525
%StandardWindowCode #GROUP	526
%StandardWindowHandling #GROUP	533
%StandardAcceptedHandling #GROUP	535
%StandardControlHandling #GROUP	536
Code Template: ControlValueValidation	539
%CodeTPLValidationCode #GROUP	541
Control Template: DOSFileLookup	543
Extension Template: DateTimeDisplay	546
%DateTimeDisplayCode #GROUP	549
PART IV—ISAM DATABASE DRIVERS	551
23 - DATABASE DRIVER OVERVIEW	553
Data Independence	553
Choosing the Right Driver	553
Common Driver Features	554
Importing File Definitions	554
Keys, Indexes, and Performance	554
Sorting and Collating Sequences	554
Debugging and Tracing File I/O	554

ERROR Messages	556
Disk Caching and Data Integrity	556
Common Driver Strings	557
ALLOWDETAILS	558
Common Driver Properties	559
PROP:SQLDriver	559
24 - ASCII DATABASE DRIVER	561
Specifications	561
Files	561
Supported Data Types	561
File Specifications/Maximums	561
Driver Strings	562
CLIP	562
CTRLZISEOF	562
ENDOFRECORD	562
FILEBUFFERS	563
TAB	563
QUICKSCAN	564
Supported Attributes and Procedures	565
25 - BASIC DATABASE DRIVER	567
Specifications	567
Files	567
Supported Data Types	567
File Specifications/Maximums	567
Driver Strings	568
ALWAYSQUOTE	568
COMMA	568
CTRLZISEOF	569
ENDOFRECORD	569
ENDOFRECORDINQUOTE	569
FIELDDELIMITER	570
FILEBUFFERS	570
QUICKSCAN	570
QUOTE	571
Popular File Formats	571

Supported Attributes and Procedures	572
26 - BTRIEVE DATABASE DRIVER	575
Specifications	575
Files	575
Data Types	576
File Specifications/Maximums	577
Driver Strings	578
ACS	578
ALLOWREAD	578
APPENDBUFFER	578
BALANCEKEYS	579
COMPRESS	579
FREESPACE	579
LACS	579
MEMO	580
PAGESIZE	581
PREALLOCATE	581
TRUNCATE	581
Driver Properties	582
PROP:PositionBlock	582
Supported Attributes and Procedures	583
Other	586
Configurable ERROR Messages	589
27 - CLARION DATABASE DRIVER	593
Specifications	593
Files	593
Data Types	593
Maximum File Specifications	594
Driver Strings	595
DELETED	595
HELD	595
IGNORECORRUPTIONS	595
IGNORESTATUS	596
MAINTAINHEADERTIME	596
RECOVER	596

Supported Attributes and Procedures	598
Other	600
Transaction Processing for Clarion Files	600
Field Labels	600
28 - CLIPPER DATABASE DRIVER	601
Specifications	601
Files	601
Data Types	601
File Specifications/Maximums	603
Driver Strings	604
BUFFERS	604
RECOVER	604
IGNORESTATUS	605
DELETED	605
Supported Attributes and Procedures	606
Other	610
Configurable ERROR Messages	613
29 - dBASEIII DATABASE DRIVER	615
Specifications	615
Files	615
Data Types	615
File Specifications/Maximums	617
Driver Strings	618
BUFFERS	618
RECOVER	618
IGNORESTATUS	619
DELETED	619
OMNIS	619
Supported Attributes and Procedures	620
Other	623
Configurable ERROR Messages	626
30 - dBASEIV DATABASE DRIVER	629
Specifications	629
Files	629

Data Types	629
File Specifications/Maximums	631
Driver Strings	632
BUFFERS	632
RECOVER	632
IGNORESTATUS	633
DELETED	633
Supported Attributes and Procedures	634
Other	637
Configurable ERROR Messages	641
31 - DOS DATABASE DRIVER	643
Specifications	643
Files	643
Data Types	644
File Specifications/Maximums	644
Driver Strings	645
FILEBUFFERS	645
QUICKSCAN	645
Supported Attributes and Procedures	646
32 - FOXPRO / FOXBASE DATABASE DRIVER	649
Specifications	649
Files	649
Data Types	649
File Specifications/Maximums	651
Driver Strings	652
BUFFERS	652
RECOVER	652
IGNORESTATUS	653
DELETED	653
Supported Attributes and Procedures	654
Other	657
Configurable ERROR Messages	661

33 - TOPSPEED DATABASE DRIVER 663

Overview 663

Specifications 664

Files 664

Data Types 664

Maximum File Specifications 664

Driver Strings 665

FLAGS 665

FULLBUILD 665

PNM= 666

TCF 666

Supported Attributes and Procedures 667

Other 670

Transaction Processing—the TCF File 673

Storing Multiple Tables in a single .TPS File 674

Collating Sequences 675

Accessing TopSpeed files with Access Jet and ODBC 676

TopSpeed Database Recovery Utility 677

ERRORCODE 90 and Corrupted Files 677

Using the Recovery Utility Interactively 678

Command Line Parameters 679

Using the Recovery Utility Non-Interactively 680

TopSpeed Database Copy Utility 682

Example Uses 682

File Sharing Considerations 682

Copy Utility Interface 683

Copy Utility Command Line Parameters 684

PART V—SQL ACCELERATOR DRIVERS 685

34 - SQL ACCELERATORS 687

Overview 687

TopSpeed's SQL Accelerators 687

Unique Keys 688

ABC Templates and SQL 688

Using SQL Tables in your Clarion Application	689
Register the SQL Accelerator	689
Import the Table Definitions	689
SQL Import Wizard—Login Dialog	690
SQL Import Wizard—Import List Dialog	690
Connection Information and Driver Configuration—File Properties	690
SQL Driver Behavior	692
Automatic Login Dialog	692
SET/NEXT and SET/PREVIOUS Processing (SELECT/ORDER BY)	692
Null Fields	693
Performance Considerations	695
Define Only the Fields You Use	695
Matching Clarion Keys to SQL Constraints and Indexes	695
ABC Template and ABC Library Optimizations	696
Inner Joins	696
Filter (Contracting) Locators	696
Approximate Record Count	696
Fixed Thumbs and Movable Thumbs	697
Batch Transaction Processing	698
Duplicated Records in BrowseBox	698
Using Embedded SQL	700
PROP:SQL	700
SEND	701
Using Embedded SQL for Batch Updates	701
Calling a Stored Procedure	702
Debugging Your SQL Application	703
System-wide Logging	703
On Demand Logging	704
Language Level Error Checking	704
SQL Driver Strings	705
ALLOWDETAILS	705
LOGFILE	705
WHERE	706
SQL Driver Properties	707
PROP:Alias	707
PROP:ConnectionString	707
PROP:Details	707

PROP:Disconnect	707
PROP:Inner	708
PROP:Log	708
PROP:LogFile	708
PROP:OrderAllTables	708
PROP:Profile	709
PROP:SQL	709
PROP:SQLFilter	709
PROP:SQLJoinExpression	710
PROP:SQLOrder	711

35 - AS400 ACCELERATOR 713

Overview	713
AS400 Server	713
AS400 Accelerator	713
Terminology	714
Installation	714
Registering the AS400 Accelerator	714
If You're New to Clarion...	716
If You're New to the AS/400...	717
System Requirements	718
Hardware	718
Software—Developer Requirements	718
Software—End User Requirements	719
Software—Client Access/400 Configuration	720
Using AS400 Accelerator	721
Importing AS/400 Files to a Data Dictionary	721
Table Names and Aliases	724
Column Names and Key Names	725
Usernames and Passwords	726
Driver Behavior	728
Logging In	728
Performance Considerations	729
Ascending and Descending Keys	729
Driver Strings	730
USETRANSACTIONS	730
GATHERATOPEN	730

Driver Properties	731
PROP:LogonScreen	731
Specifications	732
File Specification Maximums	732
Supported Data Types	732
Supported Attributes and Procedures	735
Troubleshooting	738
AS400 is not in the File Driver Drop-down List	738
Could Not Access Key Information	738
No Primary Key Found	738
Unsupported Function	739
Communication link failure	739
Windows 95	740
Cannot Load the Driver	740
Could not Logon. Specified Driver Could not be Loaded	740
CWBSY1004 Internal Error	740
Example Application	742
Running the Example Program	742
Procedures	742
36 - MSSQL ACCELERATOR	745
Overview	745
MSSQL Server	745
MSSQL Accelerator	745
SQL Import Wizard—Login Dialog	745
SQL Import Wizard—Import List Dialog	747
Connection Information and Driver Configuration—File Properties	748
Performance Considerations	750
Driver Strings	751
LOGONSCREEN	751
GATHERATOPEN	751
SAVESTOREDPROC	751
TRUSTEDCONNECTION	752
Driver Properties	753
PROP:LogonScreen	753
Supported Attributes and Procedures	754

Synchronizer Server	757
Synchronizer Login Dialog	757
37 - ODBC ACCELERATOR	761
Overview	761
What is ODBC	762
ODBC Pros and Cons	762
How ODBC Works	764
ODBC Data Types	765
Importing from ODBC Data Sources	767
Connection Information and Driver Configuration—File Properties	767
Key Configuration—Key Properties	768
Column Configuration—Field Properties	768
Debugging Your ODBC Application	769
Driver Strings	770
BINDCONSTANTS	770
CLIPSTRINGS	770
FORCEUPPERCASE	770
GATHERATOPEN	771
JOINTYPE	771
NESTING	771
ODBCCALL	771
USEINNERJOIN	772
VERIFYVIASELECT	772
ZEROISNULL	772
Driver Properties	773
PROP:AlwaysRebind	773
PROP:hdbc	773
PROP:henv	773
PROP:hstmt	773
PROP:LoginTimeout	774
PROP:LogonScreen	774
PROP:QuoteString	774
Using Embedded SQL	775
Calling a Stored Procedure	775
Supported Attributes and Procedures	776
Microsoft Access and ODBC	778

Configurable ERROR Messages	779
38 - ORACLE ACCELERATOR	781
Overview	781
Oracle Server	781
Oracle Accelerator	781
System Requirements	781
Installation	783
Registering the Oracle Accelerator	784
If You're New to Clarion...	784
If You're New to Oracle...	785
Table Import Wizard—Login Dialog	786
Table Import Wizard—Import List Dialog	788
Performance Considerations	789
Automatic Login Dialog	789
Generating Unique Key Values	790
Driver Strings	792
HINT	792
PERSONAL	793
WHERE	793
Driver Properties	794
PROP:Hint	794
Using Embedded SQL	795
PL/SQL	795
Calling a Stored Procedure	795
Supported Attributes and Procedures	796
Data Types	798
Future Oracle Releases	800
Troubleshooting	802
Clarion Won't Accept Oracle File Driver	802
Oracle Not Available (-1034)	802
Unable to allocate memory on user side (-1019)	802
Unable to spawn new ORACLE (-9352)	802
Could Not Log On	803
Invalid Field Type Descriptor	803
Unexpected End of SQL Command (-921)	803
File Not Found	803

Error 47	804
Internal Error 02: WSLDIAL	804
No Interface Driver Connected(-03121)	804
Example Application	805
Running the Example Program	805
Procedures	805
39 - SCALABLE/PERVASIVE.SQL ACCELERATOR	809
Overview	809
Scalable/Pervasive.SQL Server	809
Scalable SQL Driver	809
SQL Import Wizard—Login Dialog	809
SQL Import Wizard—Import List Dialog	811
Connection Information and Driver Configuration—File Properties	811
Performance Considerations	813
Named Databases	813
Driver Strings	814
LOGONSCREEN	814
GATHERATOPEN	814
Driver Properties	815
PROP:LogonScreen	815
Supported Attributes and Procedures	818
Synchronizer Server	820
Synchronizer Login Dialog	820
40 - SQLANYWHERE ACCELERATOR	823
Overview	823
SQLAnywhere Server	823
SQLAnywhere Accelerator	823
Start the SQLAnywhere Client	823
SQL Import Wizard—Login Dialog	824
SQL Import Wizard—Import List Dialog	826
Connection Information and Driver Configuration—File Properties	826
Driver Strings	827
LOGONSCREEN	827
GATHERATOPEN	827
Driver Properties	828

PROP:LogonScreen	828
Using Embedded SQL	829
Calling a Stored Procedure	829
Supported Attributes and Procedures	830
Synchronizer Server	832
Start the SQLAnywhere Client	832
Synchronizer Login Dialog	833
INDEX	835

FOREWORD

Welcome to the Clarion Programmer's Guide! This book is designed to provide the Clarion programmer with a lot of very useful low-level information about Clarion.

This book covers many topics internal to Clarion that are not discussed in the rest of the documentation. Why aren't they discussed? The answer is simple—most of the information presented here is on topics that become most useful when creating extensions to Clarion's standard functionality (for your own productivity, or to produce add-on products to Clarion), or that document functions that are, by default, handled for you by Clarion for Windows.

That's why we created this Programmer's Guide—just for experienced Clarion programmers who have gotten past the basics (using the Application Generator) and want to know about the more advanced aspects of Clarion programming (like writing hand-code, or multi-language programming) and are looking for new challenges.

There are three parts to the Programmer's Guide:

Part I—Clarion Language Programming

In-depth discussions of Clarion language coding techniques. This starts with a Clarion Language tutorial and goes on to include such subjects as Object Oriented Programming in Clarion.

Part II—Advanced Programming Resources

In-depth discussions of such issues as: the TopSpeed Project System (all the statements that can be used in Clarion), output from the Export Text options in the Application Generator and Dictionary Editor (.TXA/.TXD file formats), mixed-language programming (Clarion and C/C++/Modula-2), along with many more.

Included on your installation disk are a collection of useful programming files that relate to these articles, such as: the standard EQUATEs and function prototypes for Windows API calls, C++ and Modula-2 header files to access any TopSpeed file driver, and the Clarion language prototypes for many useful standard C library functions.

Part III—Template Language Programming

This is Clarion's Template Language Reference, which teaches you all the syntax of the template "meta-language." It also includes annotated examples of several of Clarion's own templates to help you begin writing your own templates.

Part IV—ISAM Database Drivers

This is Clarion's reference to all the ISAM (Indexed Sequential Access Method) file drivers.

Part V—SQL Accelerator Drivers

This is Clarion's reference to all the SQL (Structured Query Language) file drivers.

All these low-level tools allow you to "stretch" the limits of Clarion programming as far as they can go. You'll find that there are no limits (other than your own imagination). Explore the depths and have fun!

PART I

CLARION LANGUAGE PROGRAMMING

1 - PROGRAM STRUCTURE

Structured Programming

The “proper” structure of a computer program is a topic that can be the beginning of a highly charged debate. Many programmers have definite, strongly held, ideas about what constitutes “proper” structure for a program, and those ideas do not always conform to another programmer’s thoughts concerning “proper” structure. Therefore, this essay is a general discussion of the Clarion Language tools which provide the ability to construct programs in your own concept of “proper” structure.

PROCEDURES

The key to any structured programming is the ability to break your program code into separate tasks to call when needed. The Clarion Language provides a statement which allows this type of task separation: `PROCEDURE`. A `PROCEDURE` can be prototyped to `RETURN` a value, and therefore may be called as part of an expression or parameter list. A `PROCEDURE` which does not `RETURN` a value may only be called explicitly as a separate program statement—it may not be used as part of an expression or parameter list.

A `PROCEDURE` prototyped to `RETURN` a value may also be called explicitly as a separate program statement when the returned value is unimportant to the context in which it is called. Doing this will generate warnings from the compiler (which may be safely ignored) unless the `PROCEDURE`’s prototype has the `PROC` attribute.

Within a procedure you may put repetitive executable code statements into `ROUTINES`. This is useful for optimizing the size of your program code and moving explicit functionality out of the main procedure logic, making the overall logic flow easier to follow. However, `ROUTINES` are only local to the procedure and may only be used within the procedure in which it resides. Therefore, the use of `ROUTINES` is not part of this discussion.

Local Stack-Based Data Declarations

Every procedure has a data declaration section and an executable code section. The data declaration section follows the keyword `PROCEDURE` and ends with the keyword `CODE`. The executable code section follows the keyword `CODE`. Any variables or data structures declared in a procedure’s data declaration section are local to that procedure. This means they are only visible within that procedure, unless passed as a parameter to another procedure.

Variables declared locally in a procedure (without the `STATIC` attribute) are dynamically allocated memory when the procedure executes. The memory for a local variable is allocated on the program's stack unless it is larger than the stack threshold, then it is allocated on the heap (but it behaves the same as a variable allocated on the stack). When the procedure is complete and program control returns to the place from which it was called, the local variable's memory is de-allocated and returned to the program for other uses. Only variables declared local to a procedure are dynamically allocated memory on the stack.

Dynamically allocated local variables makes recursive and re-entrant procedures possible. A procedure is recursive if it calls itself within its code. Each time a procedure is recursively called, it receives a new copy of its non-static local variables. Recursion is an advanced programming technique which is useful for procedures which must be executed in successive iterations.

PROGRAM MAP

Just like a procedure, a `PROGRAM` has a data declaration section (between the keywords `PROGRAM` and `CODE`) and an executable code section (following the keyword `CODE`). All variables and data structures declared in the `PROGRAM`'s data declaration section are global (available for use anywhere in the application) and are allocated memory statically.

A program's `MAP` structure is located in the global data declaration section. This `MAP` tells the program what procedures are globally available anywhere in the application.

In a Clarion program, all procedures must be prototyped in a `MAP` structure, unless they are methods of a `CLASS` (then the `CLASS` structure itself contains the prototype). A prototype of a procedure tells the compiler the name of the procedure, and how to deal with it. For a complete description of prototypes, see *PROCEDURE Prototypes* in the *Language Reference*.

MODULE

Within a `MAP`, you may have `MODULE` structures which declare the separate source file containing the procedures prototyped in that `MODULE` structure. The `MODULE` structure is the mechanism which allows the next level of program organization: grouping procedures into separate source files.

There are many reasons to split off groups of procedures into separate source files. This is the point at which debate amongst programmers with differing viewpoints becomes highly charged. Grouping procedures which accomplish

related tasks is one approach. Grouping for optimization of compile times is another reason for grouping certain files together. Another purpose of procedure grouping is the eventual creation of a Dynamic Link Library (DLL). No matter the reason, the `MODULE` structure defines the method by which your program may reflect your structural ideas.

MEMBER

The prototypes for procedures which are defined in a source file other than the `PROGRAM` file must be declared in a `MODULE` structure within a `MAP`. The source code file specified by the `MODULE` structure must begin with a `MEMBER` statement. Any source code file beginning with a `MEMBER` statement is commonly referred to as a “MEMBER module.”

The `MEMBER` statement specifies the filename of the `PROGRAM` source file to which the `MEMBER` file belongs. The `MODULE` structure in the `MAP` points to the `MEMBER` source file, and the `MEMBER` statement within that source file points back to the `PROGRAM` source file.

For example, a source code file named `MYPROG.CLW` contains the following code:

```

PROGRAM                                !Begin global data declaration section

MAP                                    !The global MAP structure
Proc1 PROCEDURE                        !A prototype for a procedure whose
    MODULE('MYPROG2')                 ! source code is in MYPROG.CLW
Proc2 PROCEDURE                        !A separate source file, MYPROG2.CLW
    END                                ! contains another procedure
END                                    !End MODULE
                                        !End MAP

CODE                                    !Begin program executable code
    !some executable source code

Proc1 PROCEDURE                        !Begin local data declaration section
CODE                                    !Begin procedure executable code
    !some executable source code

```

A second source code file, `MYPROG2.CLW`, declared in the `MODULE` structure of the `PROGRAM MAP` in the example code above, contains the following code:

```

MEMBER('MYPROG')                      !MEMBER module of the PROGRAM in MYPROG.CLW

Proc2 PROCEDURE                        !Begin local data declaration section
CODE                                    !Begin procedure executable code
    !some executable source code

```

In this example, the program has two procedures, `Proc1` and `Proc2`. The source code for `Proc2` is in a separate file, `MYPROG2.CLW`. Therefore, the


```

        !some executable source code

Proc2  PROCEDURE          !Begin local data declaration section
CODE   !Begin procedure executable code
        !some executable source code

```

The Proc2 procedure was not declared in the PROGRAM MAP, but only in the MEMBER module's MAP. Therefore, it may be called only by other procedures within the MYPROG2.CLW MEMBER module (Proc1). Var1 was declared in the MEMBER data declaration section, therefore it may be used only by the procedures actually residing in the MEMBER module—Proc1 and Proc2.

MODULEs within MEMBER MAPs

Just as in the PROGRAM MAP, a MEMBER module's MAP may also contain MODULE structures, if the procedures prototyped are in separate source files. Any procedure which is not prototyped in the PROGRAM MAP must be prototyped in the MEMBER module MAP in which it resides. This means that identical procedure prototypes are required:

- In the MODULE structure of the MEMBER module MAP in the source code file which calls the procedure.
- In the MAP structure of the second MEMBER module, which actually contains the source code definition of the procedure.

Again using the previous examples, a file named MYPROG.CLW contains the following code:

```

PROGRAM          !Begin global data declaration section
MAP              !The global MAP structure
  MODULE('MYPROG2') !A separate source file, MYPROG2.CLW
Proc1 PROCEDURE  ! contains a procedure
  END            !End MODULE
END              !End MAP
CODE             !Begin program executable code
  !some executable source code

```

The second source code file, MYPROG2.CLW, now contains the following code:

```

MEMBER('MYPROG') !The beginning of a MEMBER module's data
                  ! declaration section
MAP              !The MEMBER MAP structure
  MODULE('MYPROG3') !A separate source file, MYPROG3.CLW
Proc2 PROCEDURE  !contains another procedure
  END            !End MODULE
END              !End MAP

```

```

Proc1  PROCEDURE      !Begin local data declaration section
      CODE            !Begin procedure executable code
Proc2  !A call to Proc2
      !some executable source code

```

A third source code file, MYPROG3.CLW, contains the following code:

```

MEMBER('MYPROG')      !The beginning of a MEMBER module's data
                      ! declaration section
MAP                   !The MEMBER MAP structure
Proc2  PROCEDURE      !An identical procedure prototype to the
                      ! prototype declared in the
                      ! MYPROG2.CLW MEMBER module MAP
END                   !End MAP

Proc2  PROCEDURE      !Begin local data declaration section
      CODE            !Begin procedure executable code
      !some executable source code

```

The procedure prototype for Proc2 in the MODULE structure of the MAP in MYPROG2.CLW is duplicated in the MAP in MYPROG3.CLW. This duplication is required for all procedures declared in a MODULE structure of a MEMBER module's MAP.

The procedure prototype for Proc2 may be placed in as many MEMBER module MAP structures (within its MODULE structure) as are needed to allow as many procedures to call it as need to. This allows you to explicitly declare and use Proc2 only in the modules that need it, and in as many modules as actually do need it.

There is an advantage to explicitly declaring all procedures in the PROGRAM in local MEMBER MAPs instead of a single global MAP. That advantage is compile time—whenever you add a procedure to a global MAP, you change the PROGRAM module, which forces a re-compilation of the entire application. By explicitly prototyping all the application's procedures in MEMBER MAPs only in the modules in which the procedures are used, you can eliminate many global re-compilations, saving you time as you incrementally text the application as it is developed.

PROCEDURE MAPs

You recall that a PROCEDURE has a data declaration section, following the keyword PROCEDURE and ending at the CODE statement. In addition to data declarations, a PROCEDURE's data declaration section may also contain its own MAP structure. A PROCEDURE's MAP may not contain any MODULE structures, and declares procedure prototypes which are available locally within the PROCEDURE. These are called Local Procedures.

The definitions of any Local Procedures must immediately follow the PROCEDURE within which they are declared (in the same source code module). The advantage of Local Procedures is that they share the local data and routines of their declaring PROCEDURE. Conceptually, a Local Procedure could be thought of as a routine which can receive parameters, return a value, and contain its own data declarations.

Using the previous example, a file named MYPROG.CLW contains the following code:

```

Proc1  PROCEDURE           !Begin Local data declaration section

  MAP                       !The local MAP structure
Proc2  PROCEDURE           ! contains a local procedure declaration
  END                       !End MAP

ProcLocalVariable LONG     !A local variable declaration

  CODE                      !Begin procedure executable code
  DO MyRoutine              !Call a ROUTINE

MyRoutine ROUTINE
  ProcLocalVariable += 1    !Increment the local variable

Proc2  PROCEDURE           !Local Procedure definition
                                ! must follow the declaring PROCEDURE in the
                                ! same source module

LocalVariable LONG        !A local variable declaration
  CODE
  LocalVariable = ProcLocalVariable
                                !Use a variable from the declaring PROCEDURE
  DO MyRoutine              !Call a ROUTINE from the declaring PROCEDURE

```

The Proc2 procedure was not declared in the PROGRAM MAP or a MEMBER MAP, but only in the Proc1 PROCEDURE MAP. Therefore, it may be called only by the Proc1 procedure. ProcLocalVariable was declared in the Proc1 PROCEDURE's data declaration section, therefore it may be used only by the declaring procedure and any Local Procedures—Proc1 and Proc2. MyRoutine ROUTINE is also available only to the declaring procedure and any Local Procedures—Proc1 and Proc2.

Summary

- With the Clarion Language you may separate tasks into procedures.
- A procedure may contain ROUTINES which optimize repetitive source code.
- A procedure may have local stack-based variables which optimize run-time memory requirements and make recursion possible.

- The PROGRAM data declaration section declares all data which is globally available.
- The PROGRAM MAP declares the prototypes for all procedures which are globally available.
- A MAP may contain a MODULE structure which points to the source code for procedures which reside in separate source files.
- The MEMBER statement points back to the PROGRAM source file.
- The MEMBER module data declaration section may declare variables and data structures which are only available to the procedures that reside in the MEMBER module.
- A MEMBER module may have its own MAP structure declaring other procedures only known locally in the MEMBER module.
- A MEMBER module's MAP may contain MODULE structures which point to procedures that reside in other MEMBER modules. The MEMBER which the MODULE points to must also contain a MAP with identical prototypes for those procedures.
- Explicitly prototyping all of an application's procedures in MEMBER module MAP structures only where actually needed can significantly reduce the number of global re-compilations of the application required during development.
- Local Procedures have the advantage of sharing their declaring procedure's local data and routines. Conceptually, this makes them function like routines which can declare data, receive parameters, and return values.

2 - EASING INTO OOP

This article was originally written as a speech delivered by Richard Taylor at TopSpeed's 1997 Developer's Conference (DevCon '97) just as Beta-2 of Clarion 4 was released. Several attendees suggested that it become part of the Clarion documentation set. Thus, you have this article, fundamentally unchanged from the speech as it was delivered.

Opening Shots

David Bayliss pretty much covered the new ABC Templates and the ABC Library in his "C4 - The Big Bang" talk yesterday, so you guys should now have a pretty good idea of how you're going to use OOP in C4. What I'm going to talk about this morning are the nuts and bolts that David used to create the ABC Library – the language syntax and concepts that are used to actually write OOP code in Clarion.

I know that some of you have had a lot of experience with Object Oriented Programming in other languages, while others here have never even *said* "OOPs" in their lives.

For those of you who do have a lot of OOP background, please bear with me because I'm going to be explaining some things that may seem pretty obvious to you. I apologize in advance if things seem a little simplistic.

To the rest of you: I only have an hour this morning, so this will be a fairly quick overview. There are many books that have been written on Object Oriented Programming, so this morning I only intend to hit these concepts "once over lightly" in relation to how we've implemented them in the Clarion language. We're having an all day class on OOP Friday, which I highly recommend to anyone who really wants to learn this stuff.

OK, let's start from a point that we all should be familiar with — Procedural code.

Procedural Code Re-Visited

We all know that a PROCEDURE has a Data Section and a Code Section.

And we all know that Local Data variables declared in a PROCEDURE are only visible within the PROCEDURE and exist only while the PROCEDURE executes. Therefore, Local Data variables are in scope only within the PROCEDURE in which they are declared.

When the PROCEDURE is called, the Local variables are automatically allocated memory on the stack. This makes them available for use in the PROCEDURE's executable code statements.

Then, when a RETURN statement executes in the PROCEDURE to return to the place from which it was called, the memory used by the Local variables is automatically de-allocated from the stack and returned to the operating system.

So we can say that the lifetime and visibility of local data in a PROCEDURE is limited to the duration of the PROCEDURE execution.

OK, so what if you wanted to have multiple CODE sections able to operate on the same set of data variables? In Clarion's Procedural code you could do it two ways:

- make the variables Global,
- or move them into a Module's Data section.

Since Module Data is a concept that's fairly unique to Clarion, we'll only discuss the Global approach this morning.

By declaring your variables Globally you end up with effectively what you want: a single set of Data variables and multiple executable CODE sections that can operate on that data. One drawback though, is that the Global Data variables are not limited in lifetime and there are various schools of programming thought that hold that the proliferation of Global Data in your programs is a bad thing.

Global data is allocated static memory. This means the variables are allocated memory when the program starts execution and only de-allocated when the program terminates. By the way, the same thing is true of Module Data, so you wouldn't buy anything by going that route except limiting the visibility to the PROCEDUREs within the same module.

OK, so you can have a single set of variables which can be referenced by multiple executable CODE sections by declaring the variables as Global Data.

What if you wanted to have multiple sets of this data that could all be referenced by the same set of multiple executable CODE sections? In other words, what if you wanted to re-use the same executable code over and over again, but using different sets of data variables each time?

Well, you *could* place all the Global data variables inside a QUEUE structure, then somehow manipulate the QUEUE entries so that you could in some way sensibly maintain which QUEUE entry it is that you're currently working on. However, this would get pretty complex pretty quickly.

There is a better way. And that way is the CLASS declaration.

CLASS Declaration

The Clarion CLASS structure allows you to declare data variables — both simple data types and reference variables — and the PROCEDURES which operate on them. This is the basic declaration structure of Object Oriented Programming in Clarion. This is the structure that lets you re-use code over and over again for different sets of data.

```
MyClass      CLASS
Property     LONG
QueProperty  &SomeQueue
Method1      PROCEDURE
Method2      PROCEDURE(LONG), LONG
            END
```

The CLASS structure is a container which holds a set of data variables and the PROCEDURES which operate on them. To use the industry-standard terms for these, the CLASS structure contains properties and methods. Properties are the data, and methods are the PROCEDURES which act on those properties. Taken together, these properties and methods form a single Object.

OK, so now we've introduced and defined in Clarion terms three standard OOP buzzwords: Properties, Methods, and Object. Let me introduce the first of the three major OOP buzzwords: Encapsulation.

Encapsulation

The CLASS structure is a container for the properties and methods. According to the Oxford English Dictionary, the word Encapsulate means: *to enclose (as) in a capsule*. Therefore, the CLASS structure *encapsulates* the properties and methods declared within it.

Now, you OOP purists out there are about to say, "Wait a minute Richard, it means a lot more than that in OOP." And you're right. In common usage for OOP, encapsulation means much more than just containerizing the properties and methods, and I'll get around to all that shortly. Just be patient.

OK, so we've now got a container for properties and methods. What good does that do us? Well, you'll notice that the Clarion keyword we're using here is CLASS and not OBJECT. The reason for that is because a CLASS declaration declares a specific type of object, and it can also declare an instance of that object type (but it doesn't have to). So here's the next ten-dollar OOP buzzword we need to learn: Instantiation.

My Oxford English Dictionary defines Instantiate as: *To represent by an instance*.

It also defines Instantiation as: *The action or fact of instantiating; representation by an instance.*

The real power of the CLASS structure is that you can have multiple *instances* of the type of object the CLASS declares. Each of these instances is allocated its own set of properties, but they all share the same set of methods that operate on those properties. That's where the real promise of OOP lies, in the fact that you only need a single set of methods to operate on the properties of any object instance of a particular CLASS. The memory for the properties of a specific object instance is allocated when that object is instantiated (a ten dollar word for created).

OK, so how do we instantiate an object? There are three ways to do it:

1. A CLASS declared without the TYPE attribute declares both a type of object and an instance of that object type.
2. A simple declaration statement with the data type being the label of a previously declared CLASS structure declares an object of the same type as the CLASS.
3. Declare a reference to the label of a previously declared CLASS structure, then use the NEW and DISPOSE procedures to instantiate and destroy the object.

That's how you instantiate objects. Now you might ask, "Why is he making such a big deal about instantiation?" Good question.

The answer has to do with memory allocation for the properties that each object contains. Remember that each object gets its own discrete set of properties when the object is instantiated, so it's important to know when these things are instantiated.

Instantiation

An object declared in your program's Global data section is instantiated for you automatically at the CODE statement which marks the beginning of the Global executable code. The object is automatically destroyed for you when you RETURN from your program to the operating system. Therefore, the lifetime and visibility of a Global object is Global.

```

PROGRAM                                !Global Data and Code
MyClass CLASS                          !Declare an object and
Property LONG                          ! a type of object
Method PROCEDURE
END
ClassA MyClass                         !Declare MyClass object
ClassB &MyClass                         !Declare MyClass reference
CODE                                    !MyClass and ClassA automagically instantiated
ClassB &= NEW(MyClass)                 !Explicitly Instantiate object
! execute some code
DISPOSE(ClassB)                        !Destroy object (required)
RETURN                                  !MyClass and ClassA automagically destroyed

```

Objects declared in any module's data section are also instantiated for you automatically at the CODE statement which marks the beginning of the Global executable code and automatically destroyed for you when you RETURN from your program to the operating system. This makes their lifetime Global, despite the fact that they are only visible to PROCEDURES within the same module.

```

MEMBER('MyApp')           !Module Data

MyClass    CLASS           !Declare an object and
Property   LONG            ! a type of object
Method     PROCEDURE
           END
ClassA     MyClass        !Declare MyClass object
ClassB     &MyClass       !Declare MyClass reference

SomeProc   PROCEDURE
  !MyClass and ClassA are instantiated and
  ! destroyed at the same time as the
  ! Global objects
  !ClassB must be explicitly instantiated
  ! and destroyed in some PROCEDURE in
  ! the module

```

Objects declared in a PROCEDURE's data section are instantiated for you automatically at the CODE statement which marks the beginning of the PROCEDURE's executable code, and automatically destroyed for you when you RETURN from the PROCEDURE. This limits their lifetime and visibility to the PROCEDURE within which they are declared.

```

SomeProc   PROCEDURE      !Local Data and Code
MyClass    CLASS          !Declare an object and
Property   LONG           ! a type of object
Method     PROCEDURE
           END
ClassA     MyClass        !Declare MyClass object
ClassB     &MyClass       !Declare MyClass reference
CODE       !MyClass and ClassA automagically instantiated
ClassB &= NEW(MyClass)    !Instantiate object
  ! execute some code
DISPOSE(ClassB)          !Destroy object (required)
RETURN         !MyClass and ClassA automagically destroyed

```

OK, all that's pretty straight forward – about what you expected, right? The ones you have to keep your eye on are the objects that you declare as reference variables. These objects are only instantiated when you execute a NEW statement, and – now here's the thing to watch out for – they are never automatically destroyed for you, you must use DISPOSE to get rid of them yourself. If you forget to do that, you can end up with a memory leak in your program.

So the question now comes up, "If each object gets its own set of properties but shares the same set of methods, how does the executable code inside the method know exactly which object's properties to affect?" Another good question.

The beginning of the answer lies in Clarion's Field Qualification syntax.

Field Qualification Syntax

The properties and methods of an object are referenced in code by prepending the name of the object to the property or method using what's come to be known as "dot syntax." This is pretty standard syntax in all the various languages that support OOP. For example, assume you have an object named "Fred" and it has a method named "Work" and two properties named "Pay" and "Hours." Outside the object itself you might call that method in an assignment as "Fred.Pay = Fred.Work(Fred.Hours)"

Now, within the methods of an object, there is no way to know exactly what the object is that is currently executing. Remember, you can have many separate instances of a specific type of object which all share the same set of methods. Therefore, within the methods of an object, instead of using the object's name, you use the word SELF.

That changes the previous assignment, when made within a method of an object, to "SELF.Pay = SELF.Work(SELF.Hours)" because the code at that point doesn't know whether its referring to the properties of the Fred object or the Barney object, or the Bruce or Richard objects. It doesn't know and doesn't care, because SELF always means "whatever the current object is."

```

PROGRAM
Employee CLASS,TYPE           !Declare object TYPE
Pay      DECIMAL(7,2)
Hours    DECIMAL(3,1)
CalcPay  PROCEDURE
Work     PROCEDURE(*DECIMAL),DECIMAL
        END
Fred     Employee             !Declare object instances
Barney   Employee
CODE
Fred.Pay = Fred.Work(Fred.Hours) !Code outside the "Fred" object

Employee.CalcPay PROCEDURE    !Method Definition
CODE
SELF.Pay = SELF.Work(SELF.Hours) !Code inside object method

```

So the answer to the question, "How does the code inside the method know exactly which object's properties to affect?" is the use of SELF as the object name within the method's code.

Constructors and Destructors

There are two more things to know about what happens at an object's Instantiation and destruction. Those two things add two more OOP buzzwords to our collection: Constructors and Destructors.

When you talk about Constructors and Destructors in other OOP languages what you're actually talking about are Automatic Constructors and

Destructors. Clarion 4 fully supports Automatic Constructors and Destructors.

An Automatic Constructor is a specific method which is automatically called for you when an object is instantiated, no matter how that object is instantiated (either declared in a data section or explicitly instantiated with the NEW procedure).

To declare an Automatic Constructor in Clarion, all you need to do is create a method named CONSTRUCT in your CLASS. The CONSTRUCT method cannot receive any parameters and cannot return a value.

The most typical usage of Automatic Constructors in other languages is to allocate memory for properties and initialize them. In Clarion, since memory is automatically allocated for variables, the most common usage would probably be to initialize properties to specific starting values and to initialize reference variable properties.

An Automatic Destructor is a specific method which is automatically called for you when an object is destroyed, no matter how that object is destroyed (either by RETURN from the limit of its local scope or explicitly destroyed with the DISPOSE procedure).

Similar to the Constructor, you declare an Automatic Destructor as a method named DESTRUCT in your CLASS. The DESTRUCT method also cannot receive any parameters and cannot return a value.

The most typical usage of Automatic Destructors in other languages is to de-allocate the memory used for properties. In Clarion, since automatically allocated memory is also automatically de-allocated, the most common usage would probably be to DISPOSE of any reference variable properties that need destruction.

```
MyClass CLASS
Property LONG
Method PROCEDURE
Construct PROCEDURE !Automatic Constructor
Destruct PROCEDURE !Automatic Destructor
END
```

There is more I need to tell you about Automatic Constructors and Destructors, but that would be getting ahead of myself, so I'll get back to this subject a little later.

More About Encapsulation

OK, I told you there was some more to say about Encapsulation, didn't I? Here it is: to many OOP purists, encapsulation is all about *hiding* the properties and methods of the CLASS from the rest of the program, not just containerizing them. This makes them "Private Property" and the Clarion language fully supports this with the PRIVATE attribute.

The PRIVATE attribute on a property declaration or a method prototype means that the property or method is visible only to the other methods within that CLASS. This “hides” them from everything outside the CLASS. Without the PRIVATE attribute, the properties and methods are “public” and the rest of your program can access them whenever the object has been instantiated and is in scope.

```

PROGRAM
MyClass CLASS
MyProperty LONG,PRIVATE !Private Property
Method PROCEDURE
MyMethod PROCEDURE,PRIVATE !Private Method
END

CODE
MyClass.MyMethod !Invalid here
MyClass.Method !Valid here
MyClass.MyProperty = 10 !Invalid here

MyClass.Method PROCEDURE
CODE
SELF.MyMethod !Valid here
SELF.MyProperty = 10 !Valid here

```

There's even more to say about Encapsulation and “public” versus PRIVATE stuff in Clarion, but I'll get to that a little later, after we've talked about the next major buzzword.

Inheritance

OK, so far we've learned some major OOP buzzwords: Properties, Methods, Objects, Encapsulation, Instantiation, Constructors and Destructors.

The next one is *Inheritance*. You all know what Inheritance is, don't you? Inheritance is where someone dies and leaves you a million dollars, right? Well, almost.

Inheritance is closely coupled to another common OOP term: to derive. My trusty Oxford English Dictionary defines derive as: *To convey from one (treated as a source) to another, as by transmission, descent, etc.; to transmit, impart, communicate, pass on, hand on.*

To derive a new CLASS, you simply name the parent CLASS as the parameter to the new CLASS statement and the new CLASS declaration starts out with everything the parent CLASS had. The difference here is that, in OOP no one has to die for the child to inherit.

This brings up a couple more common OOP terms: *Base CLASS* and *Derived CLASSES*. A Base CLASS is a CLASS which has no parameter to its CLASS statement — meaning it doesn't inherit anything — while a Derived CLASS always has a parameter naming its Parent CLASS. Notice that I did not say that the parameter to the Derived CLASS statement names a Base CLASS –

it doesn't. The parameter to a Derived CLASS statement names its Parent CLASS, which could be either a Base CLASS or another Derived CLASS. This means that you can have multiple generations of inheritance.

```

PROGRAM
MyClass    CLASS                !Declare Base Class
Property   LONG
MyProperty LONG,PRIVATE        !Private = no access in derived class
Method     PROCEDURE

END
ClassA     CLASS(MyClass)      !Declare Derived Class
Aproperty  LONG                ! which inherits
Amethod    PROCEDURE          ! MyClass.Property and
                                ! MyClass.Method

CODE
ClassA.Method                !Valid method call
ClassA.MyProperty = 10       !Invalid, not accessible outside MyClass
MyClass.Amethod              !Invalid, inheritance only is one way

```

So, back to Inheritance. A Derived CLASS inherits all the public properties and methods of its Parent CLASS. It also inherits PRIVATE properties and methods, but cannot access them in any of the Derived CLASS methods because they are truly the “Private Property” of the CLASS which declares them.

What this all means is that, when you derive a CLASS from a Parent CLASS, there is no need to duplicate all the functionality already in the Parent CLASS because the Derived CLASS inherits it all.

So why derive at all? So you can re-use your code and maintain it in only one place. Because, when properly designed, a set of CLASSES has all the most common properties and methods for objects of the same or very similar type declared in the Base CLASS. Derived CLASSES then only need to declare the properties and methods which differentiate them from any other CLASSES that might be derived from the same Parent.

As Easy As Apple Pie

One example of this type of design would be designing CLASSES for Apple Pies. The Base CLASS would contain all the standard properties common to all Apple Pies, such as Apple and Crust properties and a Bake method. However, for the individual types of Apple Pies the Derived CLASS would contain the differences between the various varieties of Apple Pies.

For example, the *Dutch ApplePie* Derived CLASS might contain a CrumbleTop property, the *Traditional American ApplePie* Derived CLASS might contain a TopCrust property, and the *Grandmas ApplePie* Derived Class might contain a CaramelTop property.

```

PROGRAM

ApplePie CLASS ,TYPE           !Declare Base Class
Apple   STRING(20)
Crust   STRING(20)
Bake    PROCEDURE
        END

Dutch   CLASS(ApplePie)        !Declare Derived Class
CrumbleTop STRING(20)
        END

American CLASS(ApplePie)      !Declare Derived Class
TopCrust STRING(20)
        END

Grandmas CLASS(ApplePie)      !Declare Derived Class
CaramelTop STRING(20)
Bake     PROCEDURE            !Overridden method
        END

```

One other thing that you can do using Inheritance is override the methods. By giving a method in a Derived CLASS exactly the same prototype (name, parameter list, and any return data type) as a Parent CLASS method, you override the method in the Derived CLASS. This allows you to provide slightly different functionality in a Derived CLASS if you need to.

Multiple Inheritance

You will notice that Clarion's Inheritance syntax only allows you to derive from a single Parent. This is called Single Inheritance. There is such a thing as Multiple Inheritance in OOP theory, but you can thank us for not giving it to you.

By not giving you Multiple Inheritance, there's another ten-dollar buzzword that you DO NOT have to learn anything about – Disambiguate! Since you don't have to know anything about it, I'm not going to define it.

For those cases where multiple inheritance would be useful, there is an OOP technique called Composition which gives you the benefits of multiple inheritance without requiring you to Disambiguate anything!

Composition

The Composition technique simply means inheriting from one CLASS and also declaring a reference to another object as a property of your Derived CLASS. Therefore, your derived CLASS inherits the properties and methods from its Parent CLASS, and also contains a reference to the second object (and implicitly, all its properties and methods) instead of inheriting them. This gives the Derived CLASS full and complete access to all the inherited properties and methods and all the properties and methods of the referenced object it contains. The ABC Library classes use Composition in several places, where it's appropriate to them.

```

PROGRAM

ApplePie CLASS ,TYPE           !Declare Base Class
Apples   STRING(20)
Crust    STRING(20)
Bake     PROCEDURE
        END

IceCream CLASS,TYPE           !Declare Base Class
Flavor   STRING(20)
Scoop    PROCEDURE
        END

AlaMode  CLASS(ApplePie)      !Composition: Derive from one CLASS
OnTheSide &IceCream           ! and contain a reference to
Serve    PROCEDURE           ! an object of another CLASS
        END

```

OK, I told you there was even more Encapsulation discussion to come, right? Here it is.

And Even More About Encapsulation

In addition to “public” and PRIVATE properties and methods, there are also PROTECTED properties and methods. You recall that the PRIVATE attribute means that the property or method is only visible within the methods of the CLASS in which the PRIVATE property or method is declared, and is not accessible by any Derived CLASS. And without the PRIVATE attribute the properties and methods are “public” and visible and available to any code, whether that code is in a CLASS method or not. Well sometimes you don’t want either private or public.

Sometimes you only want a property or method to be available within the CLASS in which it is declared, or any CLASS derived from it. That’s what the PROTECTED attribute is for.

```

PROGRAM

MyClass CLASS                 !Declare Base Class
Property LONG
MyProperty LONG,PROTECTED    !Semi-Private
Method PROCEDURE
        END

ClassA CLASS(MyClass)        !Declare Derived Class
Aproperty LONG
Amethod PROCEDURE
        END

CODE
ClassA.MyProperty = 10       !Invalid out of method

ClassA.Amethod PROCEDURE
CODE
SELF.MyProperty = 1         !Valid within method

```

PRIVATE properties and methods cannot be accessed by Derived CLASSES, but PROTECTED properties and methods can be. PROTECTED properties and methods are not visible outside the CLASS within which they are declared or any CLASS derived from that CLASS. I'd guess this makes the PROTECTED properties and methods sort of "semi-private."

And More On Constructors and Destructors

And while we're going backwards, remember a while back I said there was more to tell you about Automatic Constructors and Destructors? Well now's a good time to tell you about Inheritance and how it affects Automatic Constructors and Destructors.

What happens when a Parent CLASS has a CONSTRUCT method and the Derived CLASS also needs one? We already talked about overriding methods in Derived CLASSES, so you might guess that the Derived CLASS CONSTRUCT method would simply override the Parent CLASS method. That guess would be wrong, of course.

So why is it wrong? Because overriding the Parent CLASS Constructor might mean that code that's required to initialize inherited properties would not execute. If the inherited Parent CLASS properties were not properly initialized, you might end up with unexpected behavior in your Derived CLASS. Therefore, Automatic Constructors are not automatically overridden. Instead, by default, they all execute in order when the object is instantiated.

First, the Parent CLASS Constructor executes, then the Derived CLASS Constructor executes. They execute in the order of their derivation. Base CLASS Constructors always execute first, followed by any Derived CLASS Constructors, in the order in which they were derived. The Constructor for the most derived CLASS always executes last.

The same reasoning is true for Automatic Destructors, except the order of their execution is reversed. The most derived CLASS Destructor automatically executes first when the object is destroyed, and on down the chain of derivation until the Base CLASS Destructor executes last. In other words, LIFO: Last In First Out

This is all pretty standard. The way Automatic Constructors and Destructors work by default in the Clarion language is the same way they work in most other OOP languages. However, Clarion does give you some flexibility that some other OOP languages do not.

I told you that Automatic Constructors and Destructors are not automatically overridden in Clarion. The key word here is "automatically." You CAN override them in Clarion if you want to – something you can't do in some other OOP languages. If you add the REPLACE attribute to the prototype of your CONSTRUCT or DESTRUCT method, you are telling the compiler that you DO want to override the method.

So, what does that buy you? Suppose that for some reason, you need to initialize a variable in your Derived CLASS before the Parent CLASS Constructor executes? The only way to do that is to override the Constructor. You simply place the REPLACE attribute on the Derived CLASS CONSTRUCT method's prototype then write your code.

And now you'll ask, "But Richard, what about your great argument against automatically overriding Constructors and Destructors? Have you forgotten that?"

No, I haven't forgotten. The argument against automatically overriding Constructors and Destructors is still valid, and needs to be considered carefully when you decide to explicitly override them. But we have thought of that.

PARENT

You recall that the way to reference the current object within a method's code is to use SELF instead of the object name? Well, we've provided another tool in Clarion's OOP syntax that allows you to call a method from a Parent CLASS even when you've overridden that method.

Prepending PARENT to the method name explicitly calls the Parent CLASS's method, even though it has been overridden. This technique holds true not only for Constructors and Destructors, but also for any overridden methods.

```

PROGRAM

MyClass      CLASS,TYPE      !Declare Base Class
Property     LONG
Method       PROCEDURE
Construct    PROCEDURE
END

ClassA       CLASS(MyClass) !Declare Derived Class
Aproperty    LONG
Construct    PROCEDURE,REPLACE
END

CODE                                     !ClassA Instantiation here

ClassA.Construct PROCEDURE
CODE
  SELF.Aproperty = 1      !Initialize then call
  PARENT.Construct       ! parent constructor

```

Therefore, when you need to explicitly control the execution order of your Constructors or Destructors, you simply put the REPLACE attribute on the prototype, then in your Constructor or Destructor method's code directly call PARENT.CONSTRUCT or PARENT.DESTRUCT at the exact point within your logic that is most appropriate to what you need to do. That could be before, after, or in the middle of your Derived CLASS's code – wherever you need it to be.

Polymorphism

OK, there's one last, really major OOP buzzword you have to learn: Polymorphism. This is a really big one, and it's the key that makes Object Oriented Programming so powerful. So let's start with the Oxford English Dictionary definition of Polymorphism: *The condition or character of being polymorphous; the occurrence of something in several different forms.*

Now the Clarion language has had some forms of polymorphism for a long time already. For example, the *? and ? parameter types which indicate an unknown data type for parameter passing allow you to write polymorphic procedures. Clarion has had these since the DOS days.

In Clarion for Windows version 2.0 we introduced Procedure Overloading which allows you to create multiple procedures using the same name but taking different parameter lists – another form of polymorphism.

OK, right about now all the OOP purists in the audience are saying to themselves, “But Richard, that's got nothing to do with Polymorphism in OOP,” and they're right.

All that polymorphic functionality that we've given you in previous versions of Clarion, going all the way back to the original DOS 1.0 version, has nothing at all to do with what OOP means when it talks about Polymorphism.

Virtual Methods

There's one more common OOP term that is virtually synonymous with Polymorphism: Virtual Methods.

Now this is the point where OOP starts to get a little tricky, so you guys that are new to OOP – don't worry if you don't “get it” right away from this morning's talk. This is the part that causes people to say that it takes at least six months to really “get” OOP and how to program in it.

The good news, as you already learned yesterday from David Bayliss, is that David thoroughly understands OOP already, and he designed both the ABC Library and Templates. Which means that your Clarion 4 template generated code will actually be written for you by David Bayliss. You get all the benefits of David's thorough understanding of OOP without needing to fully understand it yourself. And for all of you, whether you already understand OOP or not, the ABC Library is an excellent example of well-designed classes that you can study and learn from.

OK, so Virtual Methods are what OOP purists are really talking about when they say Polymorphism. What's a Virtual Method?

The simple answer to that is: A Virtual Method is a method whose prototype is present in both a Parent and Derived CLASS structure, and has the VIRTUAL attribute on both.

Now you're all probably saying to yourselves, "Right Richard, thanks a lot, that tells me nothing." But actually it tells you quite a lot. It tells you that Virtual Methods must have the exact same prototype in both a Parent and Derived CLASS, and that they both must have the VIRTUAL attribute. That's the mechanics of Virtual Methods.

So, now that you've gotten the mechanics of Virtual Methods down, we can go on to the really important stuff, like – what good are they and what do they do?

You recall that a Derived CLASS object inherits the methods of its Parent CLASS? That effectively means that a Derived CLASS can "call down" to execute a Parent CLASS method.

Well, Virtual Methods are the opposite – they allow Parent CLASS methods to "call up" to execute a Derived CLASS method.

Now you're all probably saying to yourselves, "Huh? What did he just say?" so I'll repeat it. I said, Virtual Methods allow the code in Parent CLASS methods to execute Derived CLASS methods.

More Apple Pie

The easiest way to see how this works is with an example. Let's go back to our ApplePie classes. We'll add a PreparePie method to the Base CLASS because every type of Apple Pie needs preparation. Some other common tasks that the code in PreparePie will need to execute will be CreateCrust and MakeFilling, so we'll prototype methods for each of these tasks.

```

ApplePie      CLASS,TYPE
PreparePie    PROCEDURE
CreateCrust   PROCEDURE,VIRTUAL      !Virtual Methods
MakeFilling   PROCEDURE,VIRTUAL
END

Dutch         CLASS(ApplePie)
CreateCrust   PROCEDURE,VIRTUAL      !Virtual Methods
MakeFilling   PROCEDURE,VIRTUAL
END

CODE
Dutch.PreparePie          !Will call the Dutch object's Virtuals

ApplePie.PreparePie PROCEDURE
CODE
SELF.CreateCrust
SELF.MakeFilling

```

These two methods are VIRTUAL because the actual code to accomplish them will probably be a little different for each type of pie we derive.

However, we do know that these are always the steps you need to take to prepare any apple pie, so we can write that code into the Base CLASS PreparePie method and know that it won't change.

Now we'll derive the *Dutch* ApplePie CLASS. Notice that the prototypes for CreateCrust and MakeFilling are exactly the same as in the Base CLASS, including the VIRTUAL attribute.

Now, you execute the Dutch.PreparePie method. This is an inherited method, so the Derived CLASS is "calling down" to its Parent CLASS method – ApplePie.PreparePie. That means the code that executes is the code inside the ApplePie.PreparePie method.

The first statement that executes in the PreparePie method is the call to SELF.CreateCrust. So, the question is, which CreateCrust method will actually execute?

At this point, we have two defined: the one in the *ApplePie* Base CLASS, and the other in the *Dutch* Derived CLASS. The answer is, the actual method that will execute is the Dutch.CreateCrust method, because the original call to PreparePie referenced the *Dutch* object. The ApplePie Base CLASS PreparePie method is "calling up" the chain of derivation to the Derived CLASS's CreateCrust method.

OK, so now you see how it all works — Virtual Methods are sort of "reverse inheritance."

The Base CLASS code calls a method and, at the time the code is compiled, it doesn't have any idea at all what actual code will need to execute as a result of the Virtual Method call. Because of the prototype in the Base CLASS, it knows there is a method of that name that it can call to perform a specified task, it just doesn't know exactly how that code will do its task – and it doesn't care. The ApplePie Base CLASS tells the current object to create its crust and it goes away and does it.

Now, I know some you are asking yourselves, "How can this kind of code compile? I remember my compiler class in college and the compiler needs to resolve all procedure calls at compile time." Good question, And the answer is Magic! At least, that's probably what David Bayliss would tell you.

The real answer is another common OOP term: Late Binding.

Late Binding

Early binding is the kind of stuff you were taught in your compiler theory class where all procedure calls are resolved at compile time. However, since calls to Virtual Method cannot be resolved at compile time, the compiler must build what's called a "Virtual Method Table" at compile time and set the method call up for Late Binding. At run time, when the call to the Virtual

Method executes, Late Binding means that the actual method to call is determined by a lookup into the Virtual Method Table. I'm sure that all of you know a lot about lookups, since it's a pretty standard database application design technique!

Now, in many other OOP languages this Late Binding for Virtual Methods can cause a real performance hit. However, in Clarion the entire Late Binding process takes only one more extra reference at runtime than Early Binding does, so there's *virtually* no performance hit with using Virtual Methods in Clarion.

As a matter of fact, Virtual Methods in Clarion are so efficient that, when you look at the code generated by the new ABC Templates you'll find that almost all the code generated is now in Virtual Methods. This means you've probably seen the last of any Pool Limit errors!

That brings me to the last OOP issue I want to tell you about this morning — scoping. In Beta-1 you will have noticed that some of the biggest conversion problems lay in the fact that many of the template embed points were moved into Virtual Methods. This meant you had two work arounds available to you: move any procedure local data that you wanted to reference in those embed points out of the PROCEDURE itself and into Module data; or jump through some even more difficult hoops by creating reference variables for each of the variables you wanted to reference and reference assigning to them, then changing your embed code to refer to those reference variables. Needless to say, this was a lot more work than Clarion Programmers have grown accustomed to!

So, to solve that problem, this Beta-2 release of Clarion 4 introduces a new concept: Local Derived Methods.

Local Derived Methods

Local Derived Methods are declared in a Derived CLASS structure within a PROCEDURE. The Local Derived Method definitions (their executable code, that is) must immediately follow the end of the PROCEDURE in which they are declared.

So, what's the advantage you get from these things? Local Derived Methods inherit the scope of the PROCEDURE within which they are declared. That means that all the PROCEDURE's Local Data variables and ROUTINES are visible and available inside the Local Derived Methods. That eliminates all the hoops you had to jump through in Beta-1 to access your local data in those virtual embed points. And it also enables you to call the PROCEDURE's ROUTINES from within them, too, just as if your code were still within the PROCEDURE itself.

By the way, for those of you who still remember some Pascal from college, this concept is very similar to Pascal's nested procedures.

This new implementation of scoping for Local Derived Methods is what allowed David to take the OOP concept to the hilt in the ABC Templates and ABC Library. All you need to do is run the App Wizard on any dictionary then look at the generated code to see that most generated PROCEDURES now contain very little code – everything has been moved into Local Derived Methods. And you will mostly still be able to write your embed code as if it were inside the PROCEDURE itself.

Wrap Up

OK, so that's Clarion 4 OOP in a nutshell.

You've now been exposed to the three major OOP buzzwords: Encapsulation, Inheritance, and Polymorphism; and how we've implemented them in the Clarion language.

You've also heard most of the other standard OOP terms: Properties, Methods, Objects, Instantiation, Base Classes, Derived Classes, SELF, PARENT, Constructors and Destructors, Virtual Methods, Late Binding. . .

And let's not forget the one you do NOT have to learn: Disambiguate!

As I said at the beginning of this talk, there's only so much I can say in an hour, and I've already said all I can say, so now I'll just say: Thank you!

3 - OBJECT ORIENTED PROGRAMMING (OOP)

Object Overview

What are Objects?

An object is a single “thing.”

We humans tend to group what we see in the world into classes of things. We group them according to the properties and behaviors that they have in common. Therefore, things that belong to a single class have properties and behaviors which describe the type of thing it is and what it does.

We also tend to divide the classes of things we see into a tree-shaped hierarchy of classes. The hierarchy tree starts with the most general at the bottom and grows up to the most specific. Each branch of the tree is a separate class that shares the common properties of the class above it to which it is attached (its parent) because it is derived from the parent. Each derived class must also have characteristics that are unique to it that separate it from the parent and from other classes derived from the same parent.

An object is one instance of a class of things, generally from the lowest level of the hierarchy tree. It has properties and behaviors that define what it is. For example, we see two major classes of things in our world: Living Things, and Non-living Things. We can divide the Living Things class into two sub-classes: Plants and Animals. Within the Animal class we can see many sub-classes: Mammals, Birds, Fish, etc. Then these sub-classes are further sub-divided into sub-sub-classes... This sub-division of classes goes on until you get all the way to the individual object (a single “thing”).

Why Objects?

Object-oriented programming (OOP) techniques were developed to let us more closely model the programs we write on the way we look at the real world. Designing a class hierarchy to solve a real-world problem is a matter of looking at the problem from the same perspective that we look at the natural world. We start at the most abstract—general classes that have the properties and behaviors common to all members of the class—then derive from those abstract classes the specific classes that fully describe the set of individual objects in the problem.

One major benefit of object-oriented programming is extensive code re-use. Once a behavior (method) has been coded for the more general class, it never needs to be re-written for the derived classes in which it does not need to change. Therefore, it is written once and the same code is shared amongst all the objects derived from that class.

What Makes an Object?

There are three major concepts in object-orientation: encapsulation, inheritance, and polymorphism. The Clarion language contains object-oriented extensions that cover all these concepts.

Encapsulation

Encapsulation means bundling the properties of a class (its data members) together with its methods (the procedures that operate on the data members) into one coherent unit. The most important benefit of encapsulation is the ability to treat the object as a single complete entity. This lets us use the object without knowing everything about it. It also lets us change one object without affecting other unrelated objects.

The properties of an object are the things the object knows about itself, and the methods are its behaviors—the operations it can perform. Each object has its own properties unique to itself, and it shares the same methods with all the other objects of the same type (other object instances of the same class).

For example, objects that belong to a class (such as the class “normal healthy human beings”) each have their own unique set of properties (like blue eyes or brown eyes) but they all share the same basic abilities common to the class of objects to which they belong (vision).

Inheritance

Inheritance is the mechanism that allows us to build hierarchies of classes. A derived class inherits all the properties and methods of the class from which it is derived (its base class). This provides the derived class with a “starting point” of all the common properties and methods of the more general (abstract) class. Then the derived class can add the properties and methods that differentiate it from its parent.

Inheritance is also a core element of object-orientation that provides for code re-use. The code for inherited methods that are not overridden in the derived class exists only once—in the class in which they are defined.

Polymorphism

Polymorphism generically means being able to call a method that operates differently depending on how it is called. For example, the Clarion OPEN statement is polymorphic because it performs different operations depending on whether it is given a FILE or a WINDOW to open.

This type of polymorphism is generally called “Procedure overloading” because you are “overloading” what appears to be one procedure call with

multiple operations. Actually, procedure overloading is simply done in Clarion by defining separate procedures with the same name but different parameter lists. See *Procedure Overloading* in the *Language Reference* for a complete discussion of this topic.

Polymorphism in object-oriented parlance is more commonly taken to mean the ability of a base class method to call methods of classes derived from the base class—without knowing at compile-time exactly what method is actually going to be called. This is called using “Virtual Methods.” If you look at inheritance as the derived class being able to “call down” to base class methods, then you can also look at virtual methods as the base class being able to “call up” to derived class methods. This seems like a bit of magic, since the base class can never know what classes have been derived from it.

To handle virtual methods, the compiler must implement “late binding” instead of “early binding” when creating the executable. With early binding, the compiler can resolve a non-virtual procedure call at compile time to a specific code address in the executable. This means a direct call to the procedure, which is very efficient.

However, with late binding, the compiler must, at compile time, create a Virtual Method Table (VMT) that contains the specific code addresses of all the virtual methods. It then must insert code that resolves, at run time, each call to a virtual method by first looking for the method in the VMT, then calling the appropriate method. This may seem like it would create a fairly large performance hit, but with the highly efficient Clarion compiler, it is actually almost as fast as calling a non-virtual method.

This virtual method type of polymorphism is the mechanism that allows us to handle the individual differences between classes derived from the same base class while also allowing the base class to ignore those differences. This lets a common method in a base class call a method from one of its derived classes to specifically handle some operation in a manner appropriate to the specific derived class.

For example, suppose we have a Vehicle class, which contains a virtual method to Steer the vehicle, because all vehicles must have some way to steer themselves. When you derive a Bicycle class from the Vehicle class, the derived class contains its own specific Steer virtual method appropriate to a Bicycle. Then when you derive a Car class from the Vehicle class, it also contains its own specific Steer virtual method for a Car. Of course, both the Bicycle and Car derived classes also inherit a common Move method which calls the Steer method. The inherited Move method does not know or care whether it’s actually calling the Car’s Steer method or the Bicycle’s Steer method at runtime, it just tells the object (whether it’s a Car or a Bicycle) to Steer itself (and it does).

Clarion's OOP Extensions

The specific Clarion language syntax that allows you to declare classes of objects and derive classes from previously declared base classes starts with the CLASS structure.

The CLASS Structure—Encapsulation

The Clarion CLASS structure declares an object class containing properties and declaring the methods that operate on those properties. In Clarion, the properties of the class are the data members declared in the CLASS and methods are the PROCEDURES prototyped in the CLASS structure.

This example declares a very simple CLASS structure:

```
SomeClass      CLASS,MODULE('SomeClas.CLW')
PropertyA     LONG                      !Data member (property)
PropertyB     LONG
ManipulateAandB PROCEDURE                !Method
END
```

This CLASS is named **SomeClass** and it contains two properties (data members): **PropertyA** and **PropertyB**. It also contains one method: the **ManipulateAandB PROCEDURE**. The **MODULE** attribute on the CLASS statement specifies that the code defining the **ManipulateAandB PROCEDURE** exists in the **SomeClas.CLW** file.

CLASS Properties

The properties (data members) that a CLASS may contain are limited to the data types that are appropriate to have in a GROUP structure. This means that all the simple data types are allowed (LONG, SHORT, REAL, etc.) including GROUP structures, but complex data types (FILE, QUEUE, WINDOW, etc.) are not allowed. Allowing only simple data types in a CLASS may seem like a limitation, but it actually is not because Reference variables are also allowed (more on this later).

CLASS Methods

The methods (PROCEDURES) declared in a CLASS are defined in the source file named in the MODULE attribute. This encapsulates all the methods in a single source file, making maintenance much easier, and also offers some other advantages that we will get to later.

For the CLASS declaration above, the SomeClas.CLW file would contain code similar to the following:

```

SomeClass.ManipulateAandB PROCEDURE
CODE
MESSAGE('A = ' & SELF.PropertyA)
MESSAGE('B = ' & SELF.PropertyB)

```

The label of the PROCEDURE statement begins with the name of the CLASS to which the method belongs, prepended to the name of the method, and connected with a period. You can also label the method and name the CLASS to which it belongs as an implicit first parameter (and labeling the classname SELF), like this:

```

ManipulateAandB PROCEDURE(SomeClass SELF)
CODE
MESSAGE('A = ' & SELF.PropertyA)
MESSAGE('B = ' & SELF.PropertyB)

```

Within a method, you use the SELF keyword, not the CLASS label, to address a property or method of the current instance of the CLASS (the current object) on which the method is operating.

Creating Objects

An object is one specific instance of a CLASS containing its own set of properties, specific to that one instance. All object instances of a CLASS share all the methods of the CLASS, so the methods exist only once each.

A CLASS declaration with the TYPE attribute only declares the CLASS—there are no objects until you create an instance of the CLASS (also called instantiation). A CLASS declaration without the TYPE attribute both declares the CLASS and instantiates it (creates the first object).

There are two ways to create an object: declare it in a data section, or create one dynamically in executable code using NEW. To declare an object in the data section, simply name the CLASS as the data type:

```

PROGRAM
SomeClass CLASS !Creates a CLASS and an instance
PropertyA LONG
PropertyB LONG
ManipulateAandB PROCEDURE
END
MyClass CLASS,TYPE !Creates just a CLASS
PropertyA LONG
PropertyB LONG
ManipulateAandB PROCEDURE
END
SomeClassObject1 SomeClass !Create an instance of SomeClass
SomeClassObject2 SomeClass !Another instance of SomeClass
MyClassObject1 MyClass !Create an instance of MyClass
MyClassObject2 MyClass !Another instance of MyClass
CODE

```

An object declared in a data section is automatically destroyed for you when it goes out of scope. For example, an object declared in a PROCEDURE's local data section goes out of scope (and is destroyed) upon RETURN from the PROCEDURE. Objects declared in a global or module data section go out of scope (and are destroyed) only when the program terminates.

To create an object dynamically in executable code using NEW, declare a reference variable for the CLASS, then execute a reference assignment statement to the reference variable, naming the CLASS as the parameter to NEW:

```

PROGRAM
SomeClass          CLASS,TYPE
PropertyA         LONG
PropertyB         LONG
ManipulateAandB   PROCEDURE
                  END
SomeClassObjectRef &SomeClass  !A Reference to a SomeClass object
CODE
SomeClassObjectRef &= NEW(SomeClass)!Create the object

```

It is important to note that when you use NEW to instantiate the object, you must also use DISPOSE to destroy it when it goes out of scope—the object is not automatically destroyed for you. The advantage of using NEW is the ability to place the reference variable that “points at” the object inside another object as a property.

Using References as Properties

Since reference variables are valid to use anywhere the label of the data type they reference may be used, they allow a CLASS to access all the types of complex structures that cannot be directly declared inside the CLASS. When you couple with this the use of NEW and DISPOSE to dynamically create and destroy variables and objects, you give a CLASS the ability to contain almost all these complex structures.

For example, using a reference variable in the CLASS, you can declare a reference to a queue (&QUEUE) that is passed to the CLASS methods. The same thing is true of all the complex data types (&QUEUE, &FILE, &KEY, &BLOB, &VIEW, or &WINDOW). Although the CLASS cannot directly contain the data declaration of these complex structures, the CLASS methods can act as if they “own” the structure just as surely as the instance “owns” its own set of properties (data members). For example, although a FILE declaration must be external to a CLASS structure, by declaring a reference variable for the file (&FILE) within the CLASS, then executing a reference assignment statement for a specific FILE structure to an object's (one specific instance of the CLASS) reference variable, the object can “own” the FILE and its methods can execute any statement that requires the label of a FILE structure as a parameter, effectively operating directly on the FILE structure itself.

You can also declare a reference to a specific type of QUEUE, GROUP, or CLASS structure (*&QueueName*, *&GroupName*, or *&ClassName*) which a CLASS method can then dynamically create. This allows the CLASS to contain specific types of QUEUES and other named CLASSES without the need to declare them within the CLASS structure.

This example declares a simple CLASS structure containing a reference variable that “points to” a specific type of QUEUE structure:

```

MyQueue      QUEUE,TYPE      !Define a specific type of queue
Field1       LONG
Field2       STRING(20)
END
SomeClass    CLASS
QueRef       &MyQueue        !A queue with a LONG and a STRING(20)
CreateQue    PROCEDURE       !Create the queue for each object
END
SomeClass.CreateQue PROCEDURE
CODE
SELF.QueRef &= NEW(MyQueue)    !Create a QUEUE for the current object
SELF.QueRef.Field1 = 1
SELF.QueRef.Field2 = 'First Entry'
ADD(SELF.QueRef)                ! and add the first entry to the queue
IF ERRORCODE() THEN STOP(ERROR()).

```

Using these “named” references and naming a reference to a specific CLASS structure makes it possible for one object to effectively “contain” an instance of another object. This is an OOP technique known as “composition” which provides an alternative to multiple inheritance (which we will get to shortly when we discuss inheritance).

Constructors and Destructors

Constructors and destructors are a feature of many object-oriented languages. These are methods (which you must write) that execute automatically, without being explicitly called. A constructor executes when the object is created, and a destructor executes when the object is destroyed.

The most common purpose of a constructor method in other object-oriented languages is to allocate memory for the object and initialize data members to prevent bugs that could be caused by uninitialized variables. Destructor methods in other object-oriented languages usually just de-allocate the memory for the object. The Clarion language automatically allocates and de-allocates memory and initializes variables to blank, zero, or the value named in the variable’s declaration. This means the biggest reason for supporting automatic constructors and destructors doesn’t exist in Clarion. However, Clarion does support automatic constructors and destructors anyway.

If a CLASS contains a method named *Construct* (which must be a PROCEDURE that takes no parameters), then that *Construct* method is automatically called when the object is instantiated. If a CLASS contains a method named *Destruct* (also a PROCEDURE that takes no parameters), then that *Destruct* method is automatically called when the object is

destroyed. Since Clarion automatically takes care of memory allocation and initialization, the most common use of constructors and destructors in a Clarion program would be to initialize and dispose of an object's reference variables.

```

SomeClass      CLASS,TYPE
ObjectQ       &MyQueue      !Reference to a specific type of QUEUE
Construct     PROCEDURE      !Constructor method
Destruct      PROCEDURE      !Destructor method
              END
ClassRef      &SomeClass     !Reference to a SomeClass object
CODE
  ClassRef &= NEW(SomeClass)  !Create a SomeClass object
                              ! which auto-calls its constructor method
  DISPOSE(ClassRef )         !Destroy the SomeClass object
                              ! which auto-calls its destructor method

SomeClass.Construct PROCEDURE
CODE
  SELF.ObjectQ &= NEW(MyQueue) !Create the QUEUE

SomeClass.Destruct PROCEDURE
CODE
  FREE(SELF.ObjectQ)           !Free any QUEUE entires
  DISPOSE(SELF.ObjectQ)       !Destroy the QUEUE

```

In Clarion (unlike many other object-oriented languages), you may also explicitly call the constructor and destructor methods. This allows you to “re-start” an object at any time by simply calling the object's destructor (clearing it) then its constructor (re-initialize it).

Public vs. PRIVATE vs. PROTECTED

All the properties and methods of a CLASS are public unless explicitly declared with the PRIVATE or PROTECTED attribute. In this case, “public” means they are visible and available for use anywhere that the object is in scope.

PRIVATE

The PRIVATE attribute specifies that the property or method on which it is placed is visible only to the PROCEDURES defined within the source module containing the methods of the CLASS structure. This completely encapsulates the data and methods from other CLASSES.

```

SomeClass      CLASS,MODULE('SomeClass.CLW'),TYPE
PublicVar      LONG          !Declare a Public property
PrivateVar     LONG,PRIVATE  !Declare a Private property
BaseProc       PROCEDURE(REAL Parm) !Declare a Public method
Proc           PROCEDURE(REAL Parm),PRIVATE !Declare a Private method
              END

```

```

TwoClass    SomeClass                                !Declare an object
CODE
TwoClass.PublicVar = 1                               !Legal assignment
TwoClass.PrivateVar = 1                             !Illegal assignment
TwoClass.Proc(2)                                     !Illegal call to Proc

!The SomeClass.CLW source code file contains:
MEMBER('MyProg')
MAP                                                  !A local MAP which declares
SomeLocalProc  PROCEDURE(SomeClass)                ! a "friend" of SomeClass and
                                                    ! passing current object to it

END

SomeClass.BaseProc  PROCEDURE(REAL Parm)
CODE
SELF.PrivateVar = Parm                             !Legal assignment
SELF.Proc(Parm)                                     !Legal call to Proc
SomeLocalProc(SELF)
                                                    !Call the friend, passing it the current object instance

SomeClass.Proc    PROCEDURE(REAL Parm)
CODE
RETURN(Parm)

SomeLocalProc  PROCEDURE(PassedObject)             !Visible only in this module
CODE
PassedObject.PrivateVar = 1                         !Legal assignment

```

A side benefit of the Clarion implementation of **PRIVATE** properties and methods are *friends*. In C++, the concept of a *friend* is a procedure that, while not a method of a class, shares the private properties of the class. In the above example, **SomeLocalProc** is a *friend* of **OneClass** because it is defined in the same source code module as the methods that belong to **OneClass**, which gives it access to the **PRIVATE** data members. Passing the current object to **SomeLocalProc** enables it to directly address the private properties of the current object (the current instance of the **CLASS**).

PROTECTED

The **PROTECTED** attribute specifies that the property or method on which it is placed is visible only to the **PROCEDURES** defined within the source module containing the methods of the **CLASS** structure and the methods of any **CLASS** derived from that **CLASS**. This only encapsulates the data or method from other **CLASSES** not derived from the **CLASS** in which it was declared (or any other non-**CLASS** code).

The purpose of the **PROTECTED** attribute is to provide a level of encapsulation between **Public** and **PRIVATE**. **PROTECTED** data or methods are available for use within their own **CLASS** and derived **CLASSES**, but not available to any code outside those specific **CLASSES**.

Derived CLASSES—Inheritance

Clarion supports inheritance—one CLASS declaration can be built on the foundation of another CLASS's properties and methods. The CLASS from which a CLASS is derived is generally referred to as its *base class*. The derived CLASS structure inherits all the public and PROTECTED data members and methods of the base class as a starting point (but not the PRIVATE data members and methods). This means the inherited data members and methods are visible and available for use within the methods declared within the derived CLASS structure.

This example declares one CLASS structure derived from another:

```

SomeClass      CLASS                !Declare a base class
PropertyA     LONG
PropertyB     LONG
ManipulateAandB  PROCEDURE
END

AnotherClass  CLASS(SomeClass)     !Derived from SomeClass
PropertyC    LONG
ManipulateAndC  PROCEDURE
END

```

In this simple example, **AnotherClass** is derived from **SomeClass** (its base class), inheriting all the properties and methods that exist in **SomeClass**. **AnotherClass** also declares one new property and one new method that don't exist in **SomeClass**. Therefore, objects that belong to **SomeClass** have two properties and one method while objects that belong to **AnotherClass** have three properties and two methods.

Given the two CLASS declarations above, the following executable statements are all valid:

```

CODE
SomeClass.PropertyA = 10      !Assign values to the properties
SomeClass.PropertyB = 10
SomeClass.ManipulateAandB    !Call the object's method

AnotherClass.PropertyA = 10   !Assign values to the properties
AnotherClass.PropertyB = 10
AnotherClass.PropertyC = 10
AnotherClass.ManipulateAandB !Call the object's methods
AnotherClass.ManipulateAndC

```

Also given the declarations above, the following executable statements are not valid:

```

SomeClass.PropertyC = 10     !Invalid, object does not have this property
SomeClass.ManipulateAndC    !Invalid, object does not have this method

```

These statements are invalid because, although the derived object contains all the properties and methods of the CLASS from which it is derived, the reverse is not true—inheritance is one-way. There is a mechanism that gets around this and allows base class methods to call derived class methods (virtual methods) that we will get to later.

Overriding Inherited Methods

There are circumstances where a derived class may need to override an inherited method to provide the explicit functionality required for that specific class of object. This is simple to accomplish in Clarion, just re-declare the method in the derived class (with exactly the same parameter list) then re-define the method in the derived class for the new functionality.

This example overrides an inherited method:

```

SomeClass      CLASS
PropertyA     LONG
PropertyB     LONG
ManipulateAandB  PROCEDURE
                END

AnotherClass   CLASS(SomeClass)
PropertyC     LONG
ManipulateAandB  PROCEDURE           !Override the inherited method
ManipulateAndC  PROCEDURE
                END

SomeClass.ManipulateAandB  PROCEDURE
Product      LONG
                CODE
                Product = SELF.PropertyA * SELF.PropertyB
                MESSAGE('Product of A*B is ' & Product)

AnotherClass.ManipulateAandB  PROCEDURE   !Override the inherited method
DoubleProduct  LONG
                CODE
                DoubleProduct = (SELF.PropertyA * SELF.PropertyB) * 2
                MESSAGE('Double the Product of A*B is ' & DoubleProduct)

AnotherClass.ManipulateAndC  PROCEDURE
DoubleProduct  LONG
                CODE
                DoubleProduct = (SELF.PropertyA * SELF.PropertyC) * 2
                MESSAGE('Double the Product of A*C is ' & DoubleProduct)

```

In this example, **AnotherClass** overrides the **ManipulateAandB** method with its own version containing a slightly different algorithm. Notice that the parameter lists are the same (neither method receives parameters). This is the key to overriding methods in derived classes; the parameter lists must be the same.

If the parameter lists of the two ManipulateAandB methods in this example were different, then you would have procedure overloading in AnotherClass; that is, AnotherClass would actually contain two methods named ManipulateAandB and the compiler would have to resolve which one to call by means of the differing parameters. See *Function Overloading* in the *Language Reference* for a full discussion of this technique.

Simply overriding a method in a derived class does not automatically mean you cannot also call the base class method from within derived class methods. Prepending PARENT to the method call (like SELF) allows a derived class to explicitly call base class methods (PARENT.MethodName). This allows you to “incrementally” override methods without requiring that you duplicate the base class code in the derived class method that overrides it (assuming you still need the base class method’s functionality).

This example overrides an inherited method and calls the base class method from the derived class:

```
SomeClass      CLASS
PropertyA     LONG
PropertyB     LONG
ManipulateAandB  PROCEDURE
END

AnotherClass   CLASS(SomeClass)
ManipulateAandB  PROCEDURE
END

SomeClass.ManipulateAandB  PROCEDURE
Product      LONG
CODE
    Product = SELF.PropertyA * SELF.PropertyB
    MESSAGE('Product of A*B is ' & Product)

AnotherClass.ManipulateAandB  PROCEDURE
DoubleProduct  LONG
CODE
    PARENT.ManipulateAandB          !Call the PARENT method first
    DoubleProduct = (SELF.PropertyA * SELF.PropertyB) * 2
    MESSAGE('Double the Product of A*B is ' & DoubleProduct)
```

For constructor and destructor methods, simply re-declaring the *Construct* or *Destruct* method in the derived class does not override the inherited method unless the REPLACE attribute is specified on the method prototype. Without the REPLACE attribute, a derived class constructor automatically calls the base class constructor first (before the derived class constructor executes), and a derived class destructor automatically calls the base class destructor last (after the derived class destructor executes).

If the REPLACE attribute is present on the *Construct* or *Destruct* method’s prototype, then the derived class method does not automatically call the base class method. However, the base class method can be called from within the derived class method’s code by explicitly calling PARENT.*Construct* or

PARENT.Destruct. This allows you to “incrementally” customize derived class constructors or destructors without completely re-writing the base class code.

```

MyQueue      QUEUE,TYPE      !Declare a type of QUEUE with one field
Field1       STRING(10)     ! containing 10 bytes of string data
END

SomeClass    CLASS,TYPE     !Base class
ObjectQ      &MyQueue
Construct    PROCEDURE      !Constructor method
Destruct     PROCEDURE      !Destructor method
END

AnotherClass CLASS(SomeClass) !Derived class
Construct    PROCEDURE,REPLACE !Override the Constructor
END

ClassRef     &AnotherClass   !Reference to a SomeClass object
CODE
ClassRef &= NEW(AnotherClass) !Create a AnotherClass object which
                              ! calls the derived class constructor
DISPOSE(ClassRef)           !Destroy the object
                              ! which calls the base class destructor

SomeClass.Construct PROCEDURE      !Base class constructor
CODE
SELF.ObjectQ &= NEW(MyQueue)

SomeClass.Destruct PROCEDURE      !Base class destructor
CODE
FREE(SELF.ObjectQ)
DISPOSE(SELF.ObjectQ)

AnotherClass.Construct PROCEDURE    !Derived class constructor
MyString  STRING(10),DIM(10)
CODE
LOOP X# = 1 TO 10                    !Do some preliminary work, then
  MyString[X#] = 'Entry ' & FORMAT(X#,@N02)
END

PARENT.Construct                    ! call the base class constructor

LOOP X# = 1 TO 10                    ! then finish off the construction
  SELF.ObjectQ.Field1 = MyString[X#]
  ADD(SELF.ObjectQ)
  ASSERT(~ERRORCODE())
END

```

Multiple Inheritance vs. Composition

Single inheritance means that a derived class has only one base class from which it inherits properties and methods. Clarion directly supports single inheritance. Some OOP languages (most notably, C++) allow multiple inheritance, wherein a derived class inherits properties and methods from several classes. This has the advantage of easily combining existing classes to create derived classes. It also has the drawback that the compiler must deal with a lot of potential ambiguity if two or more of the classes from which the

new class is derived contain methods with the same name—you must write extra code to disambiguate the overloaded methods.

Although Clarion only supports single inheritance (like many other OOP languages) you can easily get around this limitation using a standard OOP technique called “Composition.” Composition means to place an object of one class within another. Composition provides the benefits of multiple inheritance without the potential ambiguity. For those instances where you need multiple inheritance, simply decide which class to derive from and place an object of the other class (the contained object) in the new class (the container object). You then need to implement a constructor method to instantiate the contained object.

You implement composition in Clarion by placing a Reference to the object in the CLASS declaration, like this:

```

PROGRAM
SomeClass      CLASS,TYPE
PropertyA      LONG
PropertyB      LONG
ManipulateAandB  PROCEDURE
END

AnotherClass   CLASS,TYPE
PropertyC      LONG
ManipulateAndC  PROCEDURE
END

MultiClass     CLASS(SomeClass),TYPE !Inherits from SomeClass and
AnotherClassRef &AnotherClass      ! contains an AnotherClass object
Construct      PROCEDURE           ! and a Constructor method
END

MClass         MultiClass           !Declare an object
MClassRef      &MultiClass         !Declare an object reference

CODE           !The constructor Instantiates the contained object
              ! when the container object comes into scope

MClassRef &= NEW(MultiClass)       !Create a new container object
              ! which auto-calls its constructor

MultiClass.Construct  PROCEDURE
CODE
SELF.AnotherClassRef &= NEW(AnotherClass) !Create the contained object

```

This same technique also gives you recursive classes by placing a Reference to an object of the same CLASS in the declaration, like this:

```

SomeClass      CLASS,TYPE           !SomeClass recurses into itself by
SomeClassRef   &SomeClass          ! containing a SomeClass object
PropertyA      LONG
PropertyB      LONG
ManipulateAandB  PROCEDURE
Construct      PROCEDURE
END

```

Virtual Methods—Polymorphism

Inheritance allows a derived class to “call down” to the methods it has inherited from its base class. Virtual methods, on the other hand, allow the methods in a base class to “call up” to methods in derived classes, despite not knowing exactly what is being called. To accomplish this, you must prototype the virtual method in both the base class and the derived class.

The `VIRTUAL` attribute on a method’s prototype declares a virtual method. This attribute must appear on the method’s prototype in both the base class and the derived class. Usually, the base class definition of the virtual method is a dummy procedure (one that does nothing) or one that provides some simple default functionality for those derived classes that don’t need their own method definition.

This example defines two simple virtual methods in two derived classes:

```

SomeClass      CLASS,TYPE
PropertyA     LONG
PropertyB     LONG
InitAandB     PROCEDURE(LONG PassedA, LONG PassedB)
ManipulateAandB PROCEDURE,VIRTUAL      !Declare base method virtual
END
AnotherClass  CLASS(SomeClass),TYPE
ManipulateAandB PROCEDURE,VIRTUAL      !Re-declare the virtual method
END
DifferentClass CLASS(SomeClass),TYPE
ManipulateAandB PROCEDURE,VIRTUAL      !Re-declare the virtual method
END

Object1      AnotherClass      !Declare an object
Object2      DifferentClass     !Declare a diferent object

CODE
Object1.InitAandB(10,20) !InitAandB will call the AnotherClass method
Object2.InitAandB(30,40) !InitAandB will call the DifferentClass method

SomeClass.InitAandB PROCEDURE(LONG PassedA, LONG PassedB)
CODE
SELF.PropertyA = PassedA
SELF.PropertyB = PassedB
SELF.ManipulateAandB          !Call whichever virtual method is
                              ! appropriate for the current object

SomeClass.ManipulateAandB PROCEDURE          !Base class method does
CODE                                          ! nothing

AnotherClass.ManipulateAandB PROCEDURE      !Virtual method
CODE
MESSAGE('The Product of A*B is ' & (SELF.PropertyA * SELF.PropertyB))

DifferentClass.ManipulateAandB PROCEDURE    !Virtual method
CODE
MESSAGE('The Sum of A+B is ' & (SELF.PropertyA + SELF.PropertyB))

```

In this example, the **ManipulateAandB** method is virtual. The **InitAandB** method calls **SELF.ManipulateAandB** without knowing which actual method is going to execute. When the **Object1.InitAandB(10,20)** statement executes, **SELF.ManipulateAandB** calls **AnotherClass.ManipulateAandB**, and when the **Object2.InitAandB(30,40)** statement executes, it calls the **DifferentClass.ManipulateAandB** method.

A virtual method in the derived class may explicitly call the method of the same name from the base class by calling **PARENT.VirtualMethodName**. This is just the same technique as previously demonstrated in “Overriding Inherited Methods.”

Local Derived Methods

Methods prototyped in a derived CLASS declaration within a procedure's Local data section share the declaring procedure's scope for all local data declarations and routines. The requirement for this is that the methods must be defined within the same source module as the declaring procedure and must immediately follow the procedure within that source. That is, the methods must come after any ROUTINES and before any other procedures that may be in the same source module. This means the procedure's Local data declarations and ROUTINES are all visible and can be referenced within these methods.

The most common need for this scoping is the definition of the VIRTUAL methods for a derived object declared locally to a procedure to serve some purpose for the declaring procedure. Typically, these virtual methods need access to the procedure's local data to perform their intended function.

For example:

```

MEMBER('MyApp')           !A source module
SomeClass                 CLASS,TYPE,MODULE('SomeClass.CLM')
PropertyA                 LONG
PropertyB                 LONG
InitAandB                 PROCEDURE(LONG PassedA, LONG PassedB)
ManipulateAandB           PROCEDURE,VIRTUAL
                           END

MyProc                   PROCEDURE           !Some non-object procedure

LocalVar                 LONG               !Local variable

AnotherClass             CLASS(SomeClass)    !Declare an object in local data
ManipulateAandB         PROCEDURE,VIRTUAL
                           END

CODE
DO MyRoutine
  AnotherClass.InitAandB(10,20)             !Call base class method

MyRoutine ROUTINE        !Routine local to MyProc and
  LocalVar += 10         ! AnotherClass.ManipulateAandB

```

```

AnotherClass.ManipulateAandB PROCEDURE !Virtual method with access to
X LONG ! local data and routines
CODE
LOOP 10 TIMES
  X = (SELF.PropertyA * SELF.PropertyB * LocalVar)
  MESSAGE('The Product of A*B*LocalVar is ' & X)
  DO MyRoutine !Increment the local variable
END

!MODULE('SomeClass.CLW') contains:

SomeClass.InitAandB PROCEDURE(LONG PassedA, LONG PassedB)
CODE
SELF.PropertyA = PassedA
SELF.PropertyB = PassedB
SELF.ManipulateAandB !Call whichever virtual method is
! appropriate for the current object

SomeClass.ManipulateAandB PROCEDURE !Base class method does nothing
CODE

```

Summary

- The three most important concepts in Object-orientation are: Encapsulation, Inheritance, and Polymorphism.
- The CLASS structure creates Encapsulation.
- The source code for all the methods in a single CLASS reside in a single source module, making maintenance easier.
- An object is an instance of a CLASS with its own set of data members (properties) which shares methods with all other instances of the CLASS (and also any derived CLASSES).
- A CLASS may contain data members (properties) declared as simple data types or as reference variables to complex data types (including other classes).
- Within the CLASS methods, the data members and methods appropriate to the current object instance are referenced using SELF as the object name (*SELF.DataMemberName* or *SELF.MethodName*).
- Constructor and destructor methods named *Construct* and *Destruct* are automatically called when an object is created or destroyed, and may also be explicitly called.
- All properties and methods are public unless explicitly declared with the PRIVATE or PROTECTED attribute.
- Inheritance is achieved by deriving one CLASS from another.
- You can override inherited methods by re-declaring and re-defining them in the derived CLASS, using exactly the same parameter list.

- Overridden inherited methods and VIRTUAL methods can call their base class constituents by using PARENT (just like SELF).
- Composition provides a viable alternative to multiple inheritance.
- VIRTUAL methods allow standard OOP Polymorphism, while Clarion's procedure overloading permits a non-OOP form of polymorphism.
- VIRTUAL methods are prototyped in both the base and derived classes.
- Objects local to a procedure share local variables and ROUTINEs with the declaring procedure.

4 - DATABASE DESIGN

Database Design

There are a number of methods of database organization in use today. The Inverted List Model, the Hierarchical Model, and the Network Model are three that have been widely used in the past. Mostly, these models have been used on mainframe computers, and have not been implemented on PC systems on a widespread basis. The Clarion language has the tools to allow you to utilize any of these methods, if you so choose.

By far, the most common method of database organization on PC systems today is the Relational Model, as defined by E. F. Codd. There is no database program which completely implements all of Codd's rules regarding relational database, because it is an extremely complex mathematical model. However, most database programs implement a sufficient sub-set of Codd's rules to allow practical use of the principles of the Relational Model. This essay is a very brief overview of the most fundamental aspects of relational database design as they impact business programming.

Relational Database Design

One basic principle of Relational Database involves the database design—a data item should be stored once—not duplicated in many places. There are two benefits to this: lowered disk space requirements, and easier data maintenance. To achieve this end, a relational database design splits the data into separate, related files. For example, assume a very simple order-entry system which needs to store the following data:

```
Customer Name  
Customer Address  
ShipTo Address  
Order Date  
Product Ordered  
Quantity Ordered  
Unit Price
```

This data could all be stored in each record of one file, but that would be very inefficient. The Customer Name, Address, ShipTo Address, and Order Date would be duplicated for every item ordered on every order. To eliminate the duplication, you split the data into separate files.

```
Customer File: Customer Name  
               Customer Address  
Order File:   ShipTo Address  
               Order Date  
Item File:    Product Ordered  
               Quantity Ordered  
               Unit Price
```

With this file configuration, the Customer File contains all the customer information, the Order File contains all the information that is pertinent to one order, and the Item File contains all the information for each item in the order. This certainly eliminates duplicate data. However, how do you tell which record in what file relates to what other records in which other files? This is the purpose of the relational terms “Primary Key” and “Foreign Key.”

A Primary Key is an index into a file based on a field (or fields) that cannot contain duplicate or null values. To translate this to Clarion language terms: a Primary Key would be a unique KEY (no DUP attribute) with key components that are all REQuired fields for data entry. In strict relational database design, one Primary Key is required for every file.

A Foreign Key is an index into a file based on a field (or fields) which contain values that duplicate the values contained in the Primary Key fields of another, related, file. To re-state this, a Foreign Key contains a “reference” to the Primary Key of another file.

Primary Keys and Foreign Keys form the basis of file relationships in Relational Database. The matching values contained in the Primary and Foreign Keys are the “pointers” to the related records. The Foreign Key records in “File A” point back to the Primary Key record in “File B”, and the Primary Key in “File B” points to the Foreign Key records in “File A.”

Defining the Primary and Foreign Keys for the above example requires that you add some fields to the files to fulfill the relational requirements.

```
Customer File:  Customer Number - Primary Key
                Customer Name
                Customer Address

Order File:     Order Number - Primary Key
                Customer Number - Foreign Key
                ShipTo Address
                Order Date

Item File:      Order Number - 1st Primary Key Component and Foreign Key
                Product Ordered - 2nd Primary Key Component
                Quantity Ordered
                Unit Price
```

In the Customer File, there is no guarantee that there could not be duplicate Customer Names. Therefore, the Customer Number field is added to become the Primary Key. The Order Number has been added to the Order File as the Primary Key because there is no other field that is absolutely unique in that file. The Customer Number was also added as a Foreign Key to relate the Order File to the Customer File. The Item File now contains the Order Number as a Foreign Key to relate to the Order File. It also becomes the first component of the multiple component (Order Number, Product Ordered) Primary Key.

The Relational definitions of Primary Key and Foreign Key do not necessarily require the declaration of a Clarion KEY based on the Primary or Foreign Key. This means that, despite the fact that these Keys exist in theory, you will only declare a Clarion KEY if your application actually needs it for some specific file access. Generally speaking, most all Primary Keys will have a Clarion KEY, but fewer Foreign Keys need have Clarion KEYS declared.

File Relationships

There are three types of relationships that may be defined between any two files in a relational database: One-to-One; One-to-Many (also called Parent-Child) and its reverse view, Many-to-One; and Many-to-Many. These relationships refer to the number of records in one file that are related to some number of records in the second file.

In the previous example, the relationship between the Customer File and the Order File is One-to-Many. One Customer File record may be related to multiple Order File records. The Order File and the Item File also have a One-to-Many relationship, since one Order may have multiple Items. In business database applications, One-to-Many (Parent-Child) is the most common relationship between files.

A One-to-One relationship means that exactly one record in one file may be related to exactly one record in another file. This is useful in situations where a particular file may, or may not, need to have data in some fields. If all the fields are contained in one file, you can waste a lot of disk space with empty fields in those records that don't need the extra information. Therefore, you create a second file with a One-to-One relationship to the first file, to hold the possibly unnecessary fields.

To expand the previous example, an Order may, or may not, need to have a separate ShipTo Address. So, you could add a ShipTo File to the database design.

Order File: Order Number - Primary Key
 Customer Number - Foreign Key
 Order Date

ShipTo File: Order Number - Primary Key and Foreign Key
 ShipTo Address

In this example, a record would be added to the ShipTo File only if an Order has to be shipped to some address other than the address in the Customer File. The ShipTo File has a One-to-One relationship with the Order File.

Many-to-Many is the most difficult file relationship with which to deal. It means that multiple records in one file are related to multiple records in another file. Expand the previous example to fit a manufacturing concern

which buys Parts and makes Products. One Part may be used in many different Products, and one Product could use many Parts.

```
Parts File:  Part Number - Primary Key
             Part Description
Product File: Product Number - Primary Key
             Product Description
```

Without going into the theory, let me simply state that this situation is handled by defining a third file, commonly referred to as a “Join” file. This Join file creates two One-to-Many relationships, as in this example:

```
Parts File:  Part Number - Primary Key
             Part Description

Parts2Prod File:
             Part Number - 1st Primary Key Component and Foreign Key
             Product Number - 2nd Primary Key Component and Foreign Key
             Quantity Used

Product File: Product Number - Primary Key
             Product Description
```

The Parts2Prod File has a multiple component Primary Key and two Foreign Keys. The relationship between Parts and Parts2Prod is One-to-Many, and the relationship between Product and Parts2Prod is also One-to-Many. This makes the Join file the “middle-man” between two files with a Many-to-Many relationship.

An advantage of using a Join file is that there is usually some more information that logically should be stored there. In this case, the Quantity Used (of a Part in a Product) logically only belongs in the Parts2Prod file.

Translating the Theory to Clarion

In practical relational database design, a Clarion KEY may not need to be declared for the Primary Key on some files. If there is never a need to directly access individual records from that file, then a KEY definition based on the Primary Key is not necessary. Usually, this would be the Child file (of a Parent-Child relationship) whose records are only needed in conjunction with the Parent record.

A Clarion KEY also may not need to be declared for a Foreign Key. The determination to declare a KEY is dependent upon how you are going to access the file containing the Foreign Key. If you need to access the Foreign Key records from the Primary Key, a Clarion KEY is necessary. However, if the only purpose of the Foreign Key is to ensure that the value in the Foreign Key field value is valid, no Clarion KEY is needed. Take the previous theoretical examples and create Clarion file definitions:

```

Customer      FILE,DRIVER('Clarion'),PRE(Cus)
CustKey       KEY(Cus:CustNo)           !Primary KEY
Record        RECORD
CustNo        LONG                    !Customer Number - Primary Key
Name          STRING(30)              !Customer Name
Address       STRING(30)              !Customer Address
. .

Order         FILE,DRIVER('Clarion'),PRE(Ord)
OrderKey      KEY(Ord:OrderNo)        !Primary KEY
CustKey       KEY(Ord:CustNo),DUP     !Foreign KEY
Record        RECORD
OrderNo       LONG                    !Order Number - Primary Key
CustNo        LONG                    !Customer Number - Foreign Key
Date          LONG                    !Order Date
. .

ShipTo        FILE,DRIVER('Clarion'),PRE(Shp)
OrderKey      KEY(Shp:OrderNo)        !Primary KEY
Record        RECORD
OrderNo       LONG                    !Order Number - Primary Key and Foreign Key
Address       STRING(30)              !ShipTo Address
. .

Item          FILE,DRIVER('Clarion'),PRE(Itm)
OrderKey      KEY(Itm:OrderNo,Itm:ProdNo) !Primary KEY
Record        RECORD
OrderNo       LONG                    !Order - Primary Component and Foreign Key
ProdNo        LONG                    !Prod. - Primary Component and Foreign Key
Quantity      SHORT                   !Quantity Ordered
Price         DECIMAL(7,2)            !Unit Price
. .

Product       FILE,DRIVER('Clarion'),PRE(Pro)
ProdKey       KEY(Pro:ProdNo)         !Primary KEY
Record        RECORD
ProdNo        LONG                    !Product Number - Primary Key
Description   STRING(30)              !Product Description
. .

Parts2Prod    FILE,DRIVER('Clarion'),PRE(P2P)
ProdPartKey   KEY(P2P:ProdNo,P2P:PartNo) !Primary KEY
PartProdKey   KEY(P2P:PartNo,P2P:ProdNo) !Alternate KEY
Record        RECORD
PartNo        LONG                    !Part - Primary Component and Foreign Key
ProdNo        LONG                    !Prod. - Primary Component and Foreign Key
Quantity      SHORT
. .

Parts         FILE,DRIVER('Clarion'),PRE(Par)
PartKey       KEY(Par:PartNo)         !Primary KEY
Record        RECORD
PartNo        LONG                    !Part Number - Primary Key
Description   STRING(30)              !Part Description
. .

```

Notice that only one Foreign Key (in the Order file) was explicitly declared as a Clarion KEY. A number of Foreign Keys were included as part of Primary Key declarations, but this was simply good fortune.

The Primary Key (Itm:OrderKey) defined on the Item file is there to ensure that an order does not contain duplicate Products Ordered. If this were not a consideration, Itm:OrderKey would only contain Itm:OrderNo, and would have the DUP attribute to allow duplicate KEY values. This would make it a Foreign Key instead of a Primary Key, and the file would not have a KEY defined for the Primary Key.

The Item file and the Product file have a Many-to-One relationship, which is One-to-Many looked at from the reverse perspective. This reverse view is most often used for data entry verification look-up. This means the Product Number entered into the Item file's data entry procedure can look-up and verify the Product Number against the records in the Product file.

Referential Integrity

There is one more fundamental issue in the Relational Model which should be addressed: "Referential Integrity." This is an issue which must be resolved in the executable source code for an application, because it involves the active, run-time inter-relationship of the data within the database.

Referential Integrity means that no Foreign Key can contain a value that is not matched by some Primary Key value. Maintaining Referential Integrity in your database begets two questions which must be resolved:

- What do you do when the user wants to delete the Primary Key record?
- What do you do when the user wants to change the Primary Key value?

The three most common answers to each of these questions are: **Restrict the action**, **Cascade the action**, or (less commonly) **Nullify the Foreign Key** values. Of course, there may also be application-specific answers, such as copying all information to history files before performing the action, which should be implemented as required in individual programs.

Restrict the action

Restrict the action means that when the user attempts to delete the Primary Key record, or change the Primary Key value, the action is only allowed if there are no Foreign Keys that reference that Primary Key. If related Foreign Keys do exist, the action is not allowed.

Using the files defined previously, here is an example of how the executable code might look to Restrict deletes or a change of the Primary Key value.

```

ChangeRec EQUATE(2)      !EQUATE Change Action
DeleteRec EQUATE(3)      !EQUATE Delete Action value for readability
SaveKey   LONG           !Primary Key save variable
CODE
SaveKey = Cus:CustNo      !Save Primary Key value

```

```

OPEN(window)
ACCEPT
CASE ACCEPTED()
!individual control processing
OF ?OKButton
IF Action = ChangeRec AND Cus:CustNo <> SaveKey
!Screen completion button
!Check for changed Primary Key value
Cus:CustNo = SaveKey
! change it back
MESSAGE('Key Field changes not allowed!') !tell the user
SELECT(1)
! to start over
CYCLE
ELSIF Action = DeleteRec
!Check for Delete Action
Ord:CustNo = Cus:CustNo
!Initialize Key field
GET(Order,Ord:CustKey)
! and try to get a related record
IF NOT ERRORCODE()
!If the GET was successful
MESSAGE('Delete not allowed!') ! tell user
SELECT(1)
! to start over
CYCLE
ELSE
!If GET was unsuccessful
DELETE(Customer)
! go ahead and delete it
BREAK
! and get out
END
!other executable processing statements
END
END
END
END

```

Cascade the action

Cascade the action means that when the user attempts to delete the Primary Key record, or change the Primary Key value, the action cascades to include any Foreign Keys that reference that Primary Key. If related Foreign Keys do exist, the delete action also deletes those records, and the change action also changes the values in the Foreign Keys that reference that Primary Key.

There is one consideration that should be noted when you Cascade the action. What if the file you Cascade to (the Child file) is also the Parent of another Child file? This is a situation which you must detect and handle, because the Cascade action should affect all the dependent file records. When you are writing source code to handle this situation, you need to be aware of the file relationships and write code that Cascades the action as far it needs to go to ensure that nothing is “left hanging.”

Again using the files defined previously, here is an example of how the executable code might look to Cascade deletes or a change of the Primary Key value.

```

ChangeRec EQUATE(2)      !EQUATE Change Action
DeleteRec EQUATE(3)     !EQUATE Delete Action value for readability
SaveKey LONG             !Primary Key save variable
CODE
SaveKey = Cus:CustNo     !Save Primary Key value
OPEN(window)
ACCEPT
CASE ACCEPTED()         !Process entry
!individual control processing
OF ?OKButton            !Screen completion button
IF Action = ChangeRec AND Cus:CustNo <> SaveKey
!Check for changed Primary Key value
DO ChangeCascade       ! and cascade the change
ELSIF Action = DeleteRec !Check for Delete Action
DO DeleteCascade       ! and cascade the delete
END
!other executable processing statements
END
END

ChangeCascade ROUTINE
Ord:CustNo = SaveKey     !Initialize the key field
SET(Ord:CustKey,Ord:CustKey) ! and set to process all of one
LOOP                     ! customer's orders
NEXT(Order)              ! one at a time
IF Ord:CustNo <> SaveKey OR ERRORCODE() THEN BREAK.
!Check for end of cust. and get out
Ord:CustNo = Cus:CustNo !Change to new value
PUT(Order)               ! and put the record back
IF ERRORCODE() THEN STOP(ERROR()).
END

DeleteCascade ROUTINE
Ord:CustNo = SaveKey     !Initialize the key field
SET(Ord:CustKey,Ord:CustKey) ! and set to process all of one
LOOP                     ! customer's orders
NEXT(Order)              ! one at a time
IF Ord:CustNo <> SaveKey OR ERRORCODE() THEN BREAK.
!Check for end of cust. and get out
CLEAR(Itm:Record)       !Clear the record buffer
Itm:OrderNo = Ord:OrderNo !Initialize the key field
SET(Itm:OrderKey,Itm:OrderKey) ! and set to process all of one
LOOP UNTIL EOF(Item)    ! order's items
NEXT(Item)               ! one at a time
IF Itm:OrderNo <> Ord:OrderNo OR ERRORCODE() THEN BREAK.
!Check for end of order and get out of Item loop
DELETE(Item)             ! and delete the Item record
IF ERRORCODE() THEN STOP(ERROR()).
END                       !End Item file loop
Shp:OrderNo = Ord:OrderNo !Check for ShipTo record
GET(ShipTo,Shp:OrderKey)
IF NOT ERRORCODE()      !If GET was successful
DELETE(ShipTo)          ! delete the ShipTo record
IF ERRORCODE() THEN STOP(ERROR()).
END
DELETE(Order)           ! and delete the Order record
IF ERRORCODE() THEN STOP(ERROR()).
END                       !End Order file loop

```

Nullify the Foreign Key

Nullify the Foreign Key means that when the user attempts to delete the Primary Key record, or change the Primary Key value, the Foreign Keys that reference that Primary Key are changed to null values (if the Foreign Key fields allow null values).

Again using the files defined previously, here is an example of how the executable code would look to Nullify the Foreign Keys on delete or a change of the Primary Key value.

```

ChangeRec EQUATE(2)      !EQUATE Change Action
DeleteRec EQUATE(3)      !EQUATE Delete Action value for readability
SaveKey   LONG           !Primary Key save variable
CODE
  SaveKey = Cus:CustNo    !Save Primary Key value
  OPEN(window)
  ACCEPT
  CASE ACCEPTED()        !Process entry
    !individual control processing
    OF ?OKButton         !Screen completion button
      IF Action = ChangeRec AND Cus:CustNo <> SaveKey
        !Check for changed Primary Key value
        DO ChangeNullify ! and nullify the Child records
      ELSIF Action = DeleteRec
        !Check for Delete Action
        DO DeleteNullify ! and nullify the Child records
      END
    !other executable processing statements
  END
END

ChangeNullify ROUTINE
  Ord:CustNo = SaveKey    !Initialize the key field
  SET(Ord:CustKey,Ord:CustKey) ! and set to process all of one
  LOOP                   ! customer's orders
    NEXT(Order)          ! one at a time
    IF Ord:CustNo <> SaveKey OR ERRORCODE() THEN BREAK.
    Ord:CustNo = 0       !Check for end of cust. and get out
    PUT(Order)           !Change to null value
    IF ERRORCODE() THEN STOP(ERROR()).
  END

DeleteNullify ROUTINE
  Ord:CustNo = SaveKey    !Initialize the key field
  SET(Ord:CustKey,Ord:CustKey) ! and set to process all of one
  LOOP                   ! customer's orders
    NEXT(Order)          ! one at a time
    IF Ord:CustNo <> SaveKey OR ERRORCODE() THEN BREAK.
    Ord:CustNo = 0       !Check for end of cust. and get out
    PUT(Order)           !Change to null value
    IF ERRORCODE() THEN STOP(ERROR()).
  END

```

The Nullify option does not require as many changes as the Cascade option. This is because the Cascade has to delete all the related records in as many files as are related. Nullify only needs to null out the individual Foreign Keys that reference the Primary Key being changed or deleted.

Summary

- Each data item should be stored once.
- Separate files are used to eliminate data duplication.
- Files are related by Primary and Foreign Keys.
- A Primary Key is a unique (and non-null) index into a file which provides for individual record access.
- A Foreign Key contains a reference to the Primary Key of some other file.
- One-to-Many file relationships are the most common. They are also referred to as Parent-Child and Many-to-One (same relationship, reverse view).
- One-to-One file relationships are most commonly created to hold data that is not always needed in every record.
- Many-to-Many relationships require a “Join” file which acts as a broker between the two files. The Join file inserts two One-to-Many relationships between the Many-to-Many relationship.
- Only those Primary and Foreign Keys that the application needs (as a practical consideration) for specific access to the files need to have Clarion KEYS declared.
- Referential Integrity means that all Foreign Keys contain valid references to Primary Keys.
- Maintaining Referential Integrity requires executable code that tests for Update or Delete of the Primary Key values.
- The three common solutions to maintaining Referential Integrity are: Restricting (update/delete not allowed), Cascading (also update/delete the Foreign Key), or Nullifying the Foreign Key (assign null values to the Foreign Key).

5 - DATA FILE PROCESSING

Data File Processing

Custom database applications, by definition, store data in files. Getting data into those files, and processing it for some kind of meaningful output, is the primary purpose of any database application. This essay is a discussion of the Clarion language tools which allow the programmer to access and process data files.

File Access Methods

Generally speaking, records are put into data files at the end of the file in the sequence in which they are added (this is not always true, but is usually true). This creates the “physical, record-number order” of the file—the physical order in which the records appear within the file. This physical order does not necessarily correspond to any meaningful or useful sequence.

There are two ways to access records within a file: sequential access, and random access. Sequential access means you retrieve a number of records in some specified sequence, processing each record in order. Random access means you retrieve and process one specific record. Both of these access methods are used in almost every business database application.

If you only need to access records sequentially in their physical, record-number order, nothing more than the data file is needed. If you need to randomly access a record, and you know exactly which position it occupies in the file (its record number), the same thing is true. However, for most applications, these constraints would be too limiting.

KEY and INDEX

The Clarion **KEY** and **INDEX** declarations create alternate sort orders for the records in the file. These allow sequential or random access to a data file in some order other than the physical, record-number order. The order is determined by the component fields that make up the **KEY** or **INDEX**. Each **KEY** or **INDEX** component may be in ascending or descending order.

The main difference between **KEY** and **INDEX** lies in the fact that a **KEY** is dynamically maintained. Every time a record is added, changed, or deleted, the **KEY** is also updated. Since it is always kept current, a **KEY** should be used for sort orders that are frequently used in the application.

An **INDEX** is not maintained and must be rebuilt immediately before it is used to ensure that it accurately reflects the current state of the file. The

BUILD statement is used to rebuild an INDEX. Because of the time factor in rebuilding, and the fact that exclusive file access is required for the BUILD, an INDEX should be used for sort orders that are infrequently used.

One special form of INDEX is the "dynamic" INDEX. This is an INDEX whose component fields are not declared in the file definition. The component fields of a "dynamic" INDEX are declared at run-time in the BUILD statement.

Unlike a "static" INDEX, you may BUILD a "dynamic" INDEX with the file open in any access mode. The advantage should be immediately obvious—end-user-definable sort orders.

```

Sample      FILE,DRIVER('TopSpeed'),PRE(Sam)
Field1Key   KEY(Sam:Field1)           !KEY on Field 1
Field2Ndx   INDEX(Sam:Field2)       !Static INDEX on Field2
DynNdx      INDEX()                 !Dynamic INDEX
Record      RECORD
Field1      LONG
Field2      STRING(10)
Field3      DECIMAL(7,2)
. . .

CODE
OPEN(Sample,42h)           !Open read/write deny none
LOCK(Sample)              !Lock for exclusive access
BUILD(Sam:Field2Ndx)      ! to Build the INDEX
UNLOCK(Sample)           ! then Unlock the file
BUILD(Sam:DynNdx,'-Sam:Field1,+Sam:Field2') !Build the dynamic INDEX

```

In this example, the KEY on Sam:Field1 will always be current, the INDEX on Sam:Field2 is built when the file is opened and LOCKed (exclusive access is required). The "dynamic" INDEX is built at run-time in descending Sam:Field1 and ascending Sam:Field2 sort order. Of course, it could be built in any sort order possible for the file.

Other than their maintenance, KEY and INDEX are functionally equivalent. They share the same type of file format and may be used interchangeably in all executable file access statements which require a KEY or INDEX parameter. To simplify this discussion, wherever the phrase "KEY and/or INDEX" would be appropriate, it is replaced with the generic term *index*. All references to *index* apply equally to both KEYs and INDEXes, unless otherwise noted.

Sequential File Access

There are three Clarion statements which perform sequential file access: SET, NEXT, and PREVIOUS. The SET statement initializes the sequential processing—it does not read a record. The NEXT and PREVIOUS statements read the records in ascending (NEXT) or descending (PREVIOUS) order within the sequence established by SET.

The SET statement is the ruling element in sequential file processing. A SET statement must come before NEXT or PREVIOUS to initialize the starting point and sequence in which the records will be read. Usually, the SET statement is the last executable statement before the LOOP structure which sequentially processes the records in the file. The NEXT or PREVIOUS is then the first statement within the LOOP, as in this example code using the previous example file definition:

```

SET(Sam:Field1Key)           !Set to top of file in KEY order
LOOP                         !Loop until end of the Sample file
  NEXT(Sample)               !Read each record in turn
  IF ERRORCODE() THEN BREAK. !Break at end of file
    !record processing statements
END                           !End loop

```

There are seven forms of the SET statement listed in the Language Reference Manual. These essentially break down into two categories: three starting points for physical record-number order access, and four starting points for *indexed* order access.

<u>Physical Order</u>	<u>Indexed Order</u>
Top/Bottom of File	Top/Bottom of File
Physical Record Number	Index Record Number
Index Value	Index Value
	Index Value and Physical Record Number

SET initializes the sequential processing record pointer, and it employs a type of “fuzzy logic.” When you SET to the Top/Bottom of the file, the record pointer is not actually pointing at either. If you issue a NEXT after the SET, you read records forward from the beginning of the file. If you issue a PREVIOUS instead, you read records backwards from the end of the file. Once you have issued the NEXT or PREVIOUS to begin reading records in one direction, you cannot go back across the Top/Bottom of the file without another SET.

The same “fuzzy logic” is active when you SET to an *index* value. If SET finds a record containing an exact match to that *index* value, it points to that specific record. In this case, either NEXT or PREVIOUS would read the same record.

If, however, there is no exact match to the *index* value, SET points “between” the last record in sequence containing a value less than (or greater than, in a descending *index*) the *index* value and the next record in sequence containing a value greater than (or less than, in a descending *index*) the *index* value. In this case, NEXT and PREVIOUS would not read the same record. NEXT would read the following record in the *index* sequence, PREVIOUS would read the prior record in the *index* sequence.

The advantage of this “fuzzy logic” lies in its use with a multiple component *index*, as in this example.

```

Sample      FILE,DRIVER('TopSpeed'),PRE(Sam)
FieldsKey   KEY(Sam:Field1,Sam:Field2),DUP  !KEY on Field 1 and Field 2
Record      RECORD
Field1      LONG
Field2      STRING(10)
Field3      DECIMAL(7,2)

      . .
CODE
OPEN(Sample,42h)                !Open read/write deny none
CLEAR(Sam:Record)              !Clear the record buffer
Sam:Field1 = 10                  !Initialize first KEY component
SET(Sam:FieldsKey,Sam:FieldsKey) !KEY sequence, start at 10-blank
LOOP                             !Process each record
  NEXT(Sample)                   ! one at a time
  IF ERRORCODE() THEN BREAK.     !Break at end of file
  IF Sam:Field1 <> 10             !Check for end of group
    BREAK                         ! if so, get out of process loop
  END
  !record processing statements
END                               !End process loop

```

This code first clears the record buffer, assigning zeroes to Sam:Field1 and Sam:Field3, and blanks to Sam:Field2. The first component field of Sam:FieldsKey is initialized to the value that must be in the records you need to process. The SET statement sets up sequential processing in *indexed* order, starting at the *index* value—in this case a value of 10 in Sam:Field1 and blanks in Sam:Field2.

Sample File Records:	<u>Index</u>	<u>Record #</u>	<u>Field1</u>	<u>Field2</u>	<u>Field3</u>
	1		5	ABC	14.52
	2		5	DEF	14.52
Record Pointer After SET >>	3		10	ABC	14.52
	4		10	ABC	29.04
	5		10	DEF	14.52
	6		15	ABC	14.52
	7		15	DEF	14.52

SET leaves the record pointer positioned as shown above because there is no exact match. Record 2's value 5-DEF is less than 10-blank, and record 3's value 10-ABC is greater than 10-blank, therefore the record pointer is left "between" the two. The first time through the LOOP, NEXT reads record number 3. The IF statement terminates the processing loop after NEXT reads record 6.

There is a distinct difference between the Physical Record Number and the *Index* Record Number. The Physical Record Number is the relative physical position within the data file as returned by the POINTER(Label of a File) procedure. The *Index* Record Number is the relative record position within the *index* sequence as returned by the POINTER(Label of an *Index*) procedure.

In physical order, the same file might look like this (of course, the physical and *index* record numbers are not stored in the data file):

Sample File:	Physical Record #	Index Record #	Field1	Field2	Field3
	1	3	10	ABC	14.52
	2	6	15	ABC	14.52
	3	5	10	DEF	14.52
	4	2	5	DEF	14.52
	5	4	10	ABC	29.04
	6	7	15	DEF	14.52
	7	1	5	ABC	14.52

The forms of SET that use Record Numbers as the starting point look very similar, therefore you need to be very clear about which you are using (Physical vs. *Index*).

```
SET(Sample,1)      !Physical Order, SETs to physical rec 1, index rec 3

SET(Sam:FieldsKey,1)
                  !Index order, SETs to index rec 1, physical rec 7

Sam:Field1 = 10
Sam:Field2 = 'ABC'
SET(Sam:FieldsKey,Sam:FieldsKey,5)
                  !Index order, SETs to index rec 4, physical rec 5
```

This last form of SET allows you to SET to a specific record within a sequence of records which contain duplicate *index* field values. It searches the duplicate *index* entries for an *index* entry which points to the Physical Record Number specified as the third parameter. This is useful in files where there are multiple records with duplicate *index* values and you need to begin processing at one specific record within those duplicates.

Random File Access

There is only one Clarion statement which performs random access to individual records within a file—the GET statement. Unlike SET, GET either reads the record you attempt to retrieve, or returns an error. There is no “fuzzy logic” with GET.

There are three forms of the GET statement. They allow you to retrieve a record based on an *index* value, Physical Record Number, or *Index* Record Number.

```
Sam:Field1 = 15
Sam:Field2 = 'ABC'
GET(Sample,Sam:FieldsKey)    !GETs index rec 6, physical rec 2
GET(Sample,1)                !GETs physical rec 1, index rec 3
GET(Sam:FieldsKey,1)        !GETs index rec 1, physical rec 7
```

The first GET example retrieves the first record in the *index* order which contains the values in the *index* component fields at the time the GET is issued. The second example retrieves the first record in the file in Physical

Record Number order. The third retrieves the first record in the file in *Index* Record Number order.

GET always looks for an exact match to the *index* value and returns an error if it does not find one. Therefore, all component fields of a multiple component *index* must be initialized before issuing a GET.

GET is completely independent of SET/NEXT or SET/PREVIOUS sequential processing. This means that a GET into a file which is being sequentially processed does not change the record pointer for sequential processing.

```

SET(Sam:FieldsKey)           !Set to top of file
LOOP                         !Process each record in index order
  NEXT(Sample)               !Gets each sequential record
  IF ERRORCODE() THEN BREAK. !Break at end of file
  !sequential record processing statements
  GET(Sam:FieldsKey,1)       !Gets the first record in index order
  !random access record processing statements
END

```

This example code processes through the entire file in *index* order. After each record is processed, the first record in *index* order is retrieved and processed. This does not affect the sequence, therefore NEXT will progress through the file, despite the GET of the first record every time through the loop.

Summary

- Sequential Access and Random Access are the two methods used to retrieve records from a file.
- The Clarion KEY and INDEX declarations define alternate sort orders of the file in which they are declared.
- A KEY is dynamically maintained and is always ready for use. An INDEX is not maintained and must be built before use.
- A “dynamic” INDEX allows sort orders to be defined at run-time.
- The SET statement initializes the order and starting point of sequential processing. A SET is required before the first NEXT or PREVIOUS.
- SET employs “fuzzy logic” to determine the starting point. It either points to a specific record, or “between” records at the position where it determined no record fit the starting point parameters it was given.
- Physical and *Index* Record Numbers are very different and must not be confused with each other.
- The GET statement performs random record access within a file.
 - GET is completely independent of the SET/NEXT and SET/PREVIOUS sequential record processing.

6 - MULTI-USER CONSIDERATIONS

Multi-User Considerations

The world of database applications programming is rapidly heading towards networking. Stand-alone applications are expanding into multi-user environments as more companies connect their PCs to Local Area Networks (LANs). Mainframe applications in large companies are being “right-sized” and re-written for LAN operation. With the emergence of multi-threading, multi-tasking operating systems for PCs, even standalone computers need applications that are written with multi-user shared-access considerations in mind. This essay is a discussion of the Clarion language tools provided to write applications specifically designed for use in multi-user environments.

Opening Files

Before any data file can be processed, it must first be opened. The OPEN and SHARE statements provide this function. OPEN and SHARE are functionally equivalent, the only difference between the two is the default value of the second (access mode) parameter of each.

The access mode specifies the type of access the user opening the file receives, and the type of access allowed to other users of the file. These two values are added together to create the DOS Access Code for the file. The access mode values are:

	<u>Access</u>	<u>Dec.</u>	<u>Hex.</u>
User's Access:	Read Only	0	0h
	Write Only	1	1h
	Read/Write	2	2h
Other's Access:	Deny All	16	10h
	Deny Write	32	20h
	Deny Read	48	30h
	Deny None	64	40h

The OPEN statement's default access mode is Read/Write Deny Write (22h), which only allows exclusive (single-user) disk write access to the user opening the file. The SHARE statement's default access mode is Read/Write Deny None (42h), allowing non-exclusive (multi-user) access to anybody who opens the file. Either OPEN or SHARE may open the file in any of the possible access modes.

OPEN(file)	!Open Read/Write Deny Write
OPEN(file,22h)	!Open Read/Write Deny Write
SHARE(file,22h)	!Open Read/Write Deny Write
SHARE(file)	!Open Read/Write Deny None
SHARE(file,42h)	!Open Read/Write Deny None
OPEN(file,42h)	!Open Read/Write Deny None
OPEN(file,40h)	!Open Read Only Deny None
SHARE(file,40h)	!Open Read Only Deny None

These examples demonstrate the three most commonly used access modes. For multi-user applications, the most common access mode is Read/Write Deny None (42h), which permits all users complete access to the file. Read Only Deny None (40h) is usually used in multi-user situations where the user will not update the file (a lookup-only file) but there may be some other user who may need to write to that file.

Concurrency Checking

The biggest consideration to keep in mind about multi-user access to files is the possibility that several users could be updating the same record at the same time. A process known as “concurrency checking” prevents the data file from being corrupted by multiple user updates to the same record. Concurrency checking means determining that the record on disk, which is about to be overwritten, still contains the same values it did when it was retrieved for update.

Obviously, there is no need for any kind of concurrency checking when a record is being added. If the file has a unique KEY, two users adding the same record twice is impossible because the second ADD returns a “Creates Duplicate Key” error without adding the record. If duplicate KEYS are allowed, there is no generic way for the program code to check for inadvertent (incorrect) duplicates as opposed to deliberate (correct) duplicate records. There is also no need for concurrency checking when a record is being deleted. Once the first user has deleted the record, it is gone. Any subsequent user that attempts to delete that record will not be able to get it in the first place.

Concurrency checking is necessary when a user is making a change to a record. The process of changing a record is: get the record, make the changes, and write the changes back to the file. The problem is, during the time it takes the first user to make changes to the record, a second user (a faster typist) could: get the same record, make some change, and write the changed record back to disk. When the time comes for the first user to write his/her changes to disk, the record on disk is no longer the same as when it was first retrieved. Does the first user simply overwrite the second (faster) user's changes? If both users are changing different data elements within that record and both changes are valid, overwriting the second user's changes cannot be allowed. Even if they are both making the same change, the first user needs to know that someone else has already made that change.

The simplest concurrency checking method is to execute the WATCH statement just before getting the record from disk. This tells the file driver to automatically perform concurrency checking and report an error on the PUT statement if there is a conflict. Unfortunately, not all file drivers support this.

For the simplest concurrency checking method without using WATCH, your program code should:

- 1 Save a copy of the record before any changes are made.
- 2 Re-read the record immediately before writing the changes to disk, and compare it with the saved original.
- 3 If the two are the same, allow the user's changes to be written to disk. If not, alert the user and display the record, as changed by the other user.

Assume the following global declarations and compiler equates:

```

Sample      FILE,DRIVER('TopSpeed'),PRE(Sam)      !A data file declaration
Field1Key   KEY(Sam:Field1)
Record      RECORD
Field1      LONG
Field2      STRING(10)

Action      LONG                                  !Record update action variable
AddRec      EQUATE(1)
ChangeRec   EQUATE(2)

```

Assume that some procedure allows the user to select a record from the file, defines the expected file Action (Add, Change, or Delete the record), then calls an update procedure. The update procedure operates only on that selected record and accomplishes the Action the user set in the previous procedure. The update procedure's logic would be something like this:

```

Update      PROCEDURE                            !An update procedure
Screen      WINDOW
            !data entry screen declarations go here
            END
SaveQue     QUEUE,PRE(Sav)                        !Record save queue is a copy
SaveRecord  LIKE(Sam:Record),PRE(Sav)           ! of the file's record buffer
            ! with a different prefix
SavRecPos   STRING(512)                          !Record position save variable
CODE
OPEN(Screen)
Sav:SaveRecord = Sam:Record                       !Save copy of record
ADD(SaveQue,1)                                    ! to QUEUE entry 1
SavRecPos = POSITION(Sample)                       !Save record position
DISPLAY                                           !Display the record on screen
ACCEPT                                            !Screen field process loop
CASE ACCEPTED()
    !Individual screen field edit code goes here
OF ?OKButton                                     !Screen completion field
    IF Action = ChangeRec                         !If changing an existing record
        Sav:SaveRecord = Sam:Record              !Save changes made
        ADD(SaveQue,2)                            ! to QUEUE entry 2
        GET(SaveQue,1)                            !Get original record from QUEUE
        REGET(Sample,SavRecPos)                   !Get record from FILE again
        IF ERRORCODE()
            IF ERROR() = 'RECORD NOT FOUND'      !Did someone else delete it?
                Action = AddRec                  ! change Action to add it back
                GET(SaveQue,2)                    !Get this user's changes
                Sam:Record = Sav:SaveRecord       ! put them in record buffer
            ELSE
                STOP(ERROR())                     !Stop on any other error
        END
    END

```

```

ELSIF Sav:SaveRecord <> Sam:Record !Compare for other's changes
  Sav:SaveRecord = Sam:Record      ! Save new disk record
  ADD(SaveQue,1)                   !   to QUEUE entry 1
  DISPLAY                           ! Display other's changes
  BEEP                               ! Alert the user
  MESSAGE('Changed by another station')
  SELECT(1)                         !   and start over
  CYCLE                             !   at first field
ELSE                                !If nobody changed it
  GET(SaveQue,2)                   ! Get this user's changes
  Sam:Record = Sav:SaveRecord      ! put them back in record buffer
. . .
EXECUTE Action                     !Execute disk write
  ADD(Sample)                      !If Action = 1 (AddRec)
  PUT(Sample)                      !If Action = 2 (ChangeRec)
  DELETE(Sample)                   !If Action = 3 (DeleteRec)
END
ErrorCheck                        !A generic error checking procedure
FREE(SaveQue)                     !Free memory used by queue entries
BREAK                             !   and break out of process loop
. . .                               !End loop and case

```

This example code demonstrates the simplest type of concurrency checking you can do without using WATCH. It saves the original record in memory QUEUE entry one, and the position of that record in a STRING variable. After that, the user is allowed to make the changes to the screen data. The code to check for other user's changes is contained in the CASE FIELD() OF ?OKButton. This would be the field which the user completes when he/she is done making changes and is ready to write the record to disk.

To check for other user's changes, the code first saves this user's changes to a second memory QUEUE entry, then gets the saved original record from the QUEUE. The saved record position is used to get the record from the data file again. If the record is not found in the file, someone else has deleted it. Therefore, since this user is changing it, simply add the changed record back into the file. If the record was not deleted, it is compared against the original saved copy. If they are not the same, the changed record is saved to the same memory QUEUE entry (one) which contained the original record. Then the user is alerted to the problem and sent back to the first field on the screen to re-enter the changes (if necessary). If the record is still the same, the user's changes are retrieved from the second memory QUEUE entry and put into the record buffer for the disk write. This method is fairly straight-forward and logical. However, it uses three extra chunks of memory the size of the record buffer: the memory QUEUE's buffer, and the two entries in that QUEUE (plus each QUEUE entry's 28-byte overhead). If you are dealing with a file that has many fields, the record buffer could be very large and this could use a significant amount of memory.

Another method of concurrency checking does not copy and save the original record, but instead calculates a Checksum or Cyclical Redundancy Check (CRC) value. The calculation is performed on the record before changes are made, then the record is retrieved from disk and the calculation is performed again. If the two values are not the same, the record has been changed. This method still requires a save area for the user's changes,

because the record must be read again for the second calculation, and all disk reads are placed in the record buffer. Without a save area, the user's changes would be overwritten.

Here is an example of a 16-bit CRC procedure, and its prototype for the MAP structure. This is similar to the CRC calculations used in some serial communications protocols. An array of BYTE fields is passed to the procedure, it calculates a 16-bit CRC value for that array, and returns it to a USHORT (16-bit unsigned) variable.

```

MAP                                !The procedure prototype for the MAP.
  CRC16(*BYTE[]),USHORT           !CRC16 expects an array of BYTES to be
END                                ! passed to it, returns a USHORT value
!~~~~~
CRC16  PROCEDURE(Array)           !16 Bit CRC Check
CRC    ULONG                      !Work variable
CODE
  LOOP X# = 1 TO MAXIMUM(Array,1) !Loop through whole array
    CRC = BOR(CRC,Array[X#])      !Concatenate an array byte to CRC
  LOOP 8 TIMES                    !Loop through each bit
    CRC = BSHIFT(CRC,1)          !Shift CRC left one bit
    IF BAND(CRC,1000000h)        !Was CRC 24th bit on before shift?
      CRC = BXOR(CRC,102100h)    ! XOR shifted value with CRC mask
  . . . !End both loops
  RETURN(BAND(BSHIFT(CRC,-8),0000FFFFh)) !Shift and mask return value

```

Using this CRC check procedure, the previous example code would be changed to look like:

```

Update      PROCEDURE              !An update procedure
Screen      WINDOW
            !data entry screen declarations go here
            END
Sav:SaveRecord LIKE(Sam:Record),PRE(Sav),STATIC
            !Record buffer save area
            ! with a different prefix
PassArray   BYTE,DIM(SIZE(Sam:Record),OVER(Sam:Record)
            !Declare array OVER Sam:Record
SavRecPos   STRING(512)            !Record position save variable
SavCRC      USHORT                 !CRC value save variable
CODE
  OPEN(Screen)
  SavCRC = CRC16(PassArray)         !Save original CRC value
  SavRecPos = POSITION(Sample)       !Save record position
  DISPLAY                                       !Display the record on screen
  ACCEPT                                       !Screen field process loop
  CASE ACCEPTED()
    !Individual screen field edit code goes here
  OF ?OKButton                               !Screen completion field
    IF Action = ChangeRec                !If changing an existing record
      Sav:SaveRecord = Sam:Record        !Save changes made
      REGET(Sample,SavRecPos)           !Get record from FILE again
      IF ERRORCODE()
        IF ERROR() = 'RECORD NOT FOUND' !Did someone else delete it?
          Action = AddRec                ! change Action to add it back
          Sam:Record = Sav:SaveRecord    ! put them in record buffer

```

```

ELSE
    STOP(ERROR())                !Stop on any other error
END
ELSIF SavCRC <> CRC16(PassArray)!Compare CRCs for changes
    SavCRC = CRC16(PassArray)    ! Save new CRC value
    DISPLAY                      ! Display other's changes
    BEEP                          ! Alert the user
    IF MESSAGE('Changed by another station').
    SELECT(1)                      ! and start over
    CYCLE                          ! at first field
ELSE                               !If nobody changed it
    Sam:Record = Sav:SaveRecord    ! put changes back in buffer
. .
EXECUTE Action                      !Execute disk write
ADD(Sample)                          !If Action = 1 (AddRec)
PUT(Sample)                           !If Action = 2 (ChangeRec)
DELETE(Sample)                         !If Action = 3 (DeleteRec)
END
ErrorCheck                          !A generic error checking procedure
BREAK                                ! and break out of process loop
. .                                  !End loop and case

```

You can see that the update procedure code using this method is a bit smaller, and easier to follow logically. There are two new data declarations: `SavCRC` is declared to save the original CRC calculation, and `PassArray` is an array declared **OVER** the file's record buffer. The `PassArray` declaration simply provides a way to pass the `CRC16` procedure the entire record as an array of `BYTES`, it does not allocate any memory.

A valid question at this point would be, “Why is the `Sav:SaveRecord` declared in this code with a `STATIC` attribute?” There are four reasons:

- A save area is still needed to temporarily keep the user's changes while concurrency checking calculations are being performed.
- Variables declared locally in a `PROCEDURE` are assigned memory on the stack when the `PROCEDURE` is called.
- The save area for a large record buffer could easily take more memory than is available on the stack. Therefore, the save area should be in static memory.
- In a `PROCEDURE`, only data structures (`WINDOW`, `REPORT`, `FILE`, `VIEW`, and `QUEUE`) and variables with the `STATIC` attribute are assigned static memory.

Of course, if the save area were declared in the global data section (between the keywords `PROGRAM` and `CODE`), or a `MEMBER` module's data section (between the keywords `MEMBER` and `PROCEDURE`), it would not need to be declared with the `STATIC` attribute—it would automatically be assigned static memory.

HOLD and RELEASE

A tool to prevent other users from making changes to a record while it is being updated is the HOLD statement. HOLD tells the following GET, NEXT, or PREVIOUS statement to get the record and set a flag that tells any other user attempting to get that record that it is in use—a "record lock." The record remains held until it is: explicitly released with a RELEASE statement; implicitly released by a PUT, or DELETE, of that record; or, implicitly released by retrieving another record from the same file.

The Clarion language supports multiple file systems through its file driver technology. Each file system may implement record locking in a different manner. Therefore, the actual effect of HOLD is dependent upon the file driver, which takes whatever action is appropriate to the file system. In some file systems, a HOLD on a record allows other users to read the record, but not to write to it. In others, HOLD blocks other users from any access to the record. Some file systems release the HOLD automatically if the system crashes, others don't and leave it flagged as held. The specific action of HOLD is described in each file driver's documentation.

If you HOLD a record when it is retrieved, and RELEASE it when you write it back, you can eliminate the need for the type of concurrency checking previously described. Is this a good idea, though? Depending upon the actual implementation of HOLD in the file system being used, the answer may be either Yes or No.

If your file system blocks other users from all access to the record, or there is the possibility a "system crash" could leave it in a HOLD state, the answer is probably No. This does not mean that you should not use HOLD at all. It does mean that you should not use HOLD where a record would be held during user input (an indeterminate amount of time). More likely, you would use HOLD during the concurrency check described above. This is to make sure that nobody changes the record between the second GET (for concurrency checking) and the PUT that writes the user's changes to disk.

Using HOLD in this manner only changes the code in the CASE ACCEPTED() OF ?OKButton.

```

OF ?OKButton                                !Screen completion field
  IF Action = ChangeRec                      !If changing an existing record
    Sav:SaveRecord = Sam:Record             !Save changes made
    HOLD(Sample,1)                          !Hold while checking for changes
    REGET(Sample,SavRecPos)                 !Get record from FILE again
    IF ERRORCODE()
      IF ERROR() = 'RECORD ALREADY HELD' !Has someone else got it?
        BEEP                                ! Alert the user
        SHOW(25,1,'Held by another station')
        SELECT(1)                           ! and start over
        CYCLE                                ! at first field
      ELSIF ERROR() = 'RECORD NOT FOUND' !Did someone else delete it?
        Action = AddRec                     ! change Action to add it back

```

```

        Sam:Record = Sav:SaveRecord ! put them back in record buffer
    ELSE
        STOP(ERROR())              !Stop on any other error
    END
    ELSIF SavCRC <> CRC16(PassArray) !Compare CRC values for changes
        RELEASE(Sample)            !Release the hold
        SavCRC = CRC16(PassArray)  ! Save new CRC value
        DISPLAY                    ! Display other's changes
        BEEP                       ! Alert the user
        IF MESSAGE('Changed by another station').
        SELECT(1)                   ! and start over
        CYCLE                       ! at first field
    ELSE
        Sam:Record = Sav:SaveRecord !If nobody changed it
        Sam:Record = Sav:SaveRecord ! put these changes back in buffer
    .
EXECUTE Action                    !Execute disk write
ADD(Sample)                       !If Action = 1 (AddRec)
PUT(Sample)                       !If Action = 2 (ChangeRec)
DELETE(Sample)                   !If Action = 3 (DeleteRec)
END
ErrorCheck                       !A generic error checking procedure
BREAK                             ! and break out of process loop

```

This change puts a **HOLD** on the record only long enough to determine if it is the same record the user started with and write the changes to disk. If someone else has a **HOLD** on the record, the user is alerted to that fact and allowed to try again. If the record continually comes up as held by another station, then it has probably been left in a **HOLD** state by a system crash. In that case, the hold should be released by whatever action is appropriate for that file system. Code could be written to handle this eventuality, but it would be specific to the file system and this is “generic” example code.

If the prevention of record update conflicts is a “mission-critical” concern, then **HOLD** could be used to keep control of the record during user data entry. One trade-off with this use of **HOLD** is the nuisance of dealing with records that are left locked when users’ systems crash while records are held. Correcting that situation could involve some manual work with file system utilities, or could simply be a matter of specific coding considerations for the file system being used. Another concern with using **HOLD** this way comes when the file system being used does not allow other users to read the held records. The held records would seem to “disappear” then “reappear” from time to time as users **HOLD** records. Either way, this method should probably not be used unless the application really requires it.

To utilize this technique, the **HOLD** would have to be in the procedure which actually retrieves the record from the file. In most cases, that procedure would display some kind of scrolling list of records, usually displayed in a **LIST** box. The following example code demonstrates this.

```

OF ?List                          !LIST
CASE EVENT()
OF EVENT:Accepted                 !An existing record was selected
    GET(TableQue,CHOICE())        !Get record number from the QUEUE

```

```

HOLD(Sample,1)                !Arm the HOLD
REGET(Sample,Que:RecPosition) ! and get the record from the file
IF ERROR() = 'RECORD ALREADY HELD' !Has someone else got it?
  BEEP                          ! Alert the user
  IF MESSAGE('Held by another station').
  SELECT(?List)                 ! to try again
  CYCLE
ELSE                             !If no one else has it
  Action = ChangeRec           !Set up disk action for change
  Update                       ! and call the update procedure
END
!Code to handle other keycodes goes here
END

```

This technique grossly simplifies the update procedure code, as in this example:

```

Update PROCEDURE !An update procedure
Screen WINDOW
      !data entry controls go here
      END
CODE
OPEN(Screen)
DISPLAY                               !Display the record on screen
ACCEPT                                !Screen field process loop
  CASE FIELD()
    !Individual screen field edit code goes here
  OF ?OKButton                         !Screen completion field
    CASE EVENT()
      OF EVENT:Accepted
        EXECUTE Action                 !Execute disk write
          ADD(Sample)                   !If Action = 1 (AddRec)
          PUT(Sample)                   !If Action = 2 (ChangeRec)
          DELETE(Sample)                !If Action = 3 (DeleteRec)
        END
      ErrorCheck                       !A generic error checking procedure
      BREAK                             ! and break out of process loop
    . . .                               !End loop and case

```

The HOLD statement only allows each user to HOLD one record in each file. If you need to update multiple records in one file and you must be sure that no other user makes changes to those records while they are being updated, then you must LOCK the file.

LOCK and UNLOCK

The LOCK statement prevents other users from accessing any records in a file, until you UNLOCK it. Just like HOLD, the effect of LOCK is dependent upon the file driver which takes whatever action is appropriate to the file system. In some file systems, a system crash automatically unlocks the file, and in others it is left locked. The specific action LOCK takes is described in each file driver's documentation.

Because other users are completely barred from accessing records in the LOCKed file, LOCK is not commonly used. The most common use of LOCK would be to BUILD an INDEX prior to using it (and that is not even necessary if it is a “dynamic” INDEX). The type of “batch update processing” that would require a file to be LOCKed for a significant period of time is generally best left until after hours, when all users are gone. Other than to BUILD an INDEX, a file LOCK is usually only needed during Transaction Processing, which is the subject of a separate essay.

If an application truly demands a file LOCK, then the period of time during which the other users are denied access should be kept to an absolute minimum. The code between the file LOCK and its subsequent UNLOCK statement should not require any user input. This means that the code should be written such that an end-user cannot go to lunch leaving a file LOCKed. Specifically, LOCK should come immediately before the BUILD occurs, and the file should be UNLOCKed as soon as it is complete.

```
ReportProc      PROCEDURE
Sample         FILE,DRIVER('TopSpeed'),PRE(Sam) !A data file declaration
Field1Key      KEY(Sam:Field1)
Field2Ndx      INDEX(Sam:Field2) !An INDEX
Record         RECORD
Field1         LONG
Field2         STRING(10)
. . .
Report         REPORT
               !Report declaration statements go here
               END

CODE
OPEN(Sample,42h) !Open Read/Write Deny None
LOCK(Sample,1)  !Lock the file
IF ERROR() = 'FILE IS ALREADY LOCKED' !Check for other locks
  BEEP !Alert the user
  IF MESSAGE('Locked by another station').
  RETURN ! and get out
END
BUILD(Sam:Field2Ndx) !Build the index
UNLOCK(Sample) !Unlock the file
OPEN(Report)
SET(Sam:Field2Ndx) !Use the index
LOOP
  NEXT(Sample)
  IF ERRORCODE() THEN BREAK.
  !Report processing code goes here
END
```

This code opens the file in *access mode* 42h (Read/Write Deny None) for fully shared access. The LOCK is attempted for one second. If it is successful, the BUILD immediately executes. If the LOCK was unsuccessful, the user is alerted and returned to the procedure that called the report. Once the BUILD is complete, UNLOCK once again allows other users access to the file, and the report is run based on the sort order of the INDEX.

”Deadly Embrace”

There are two forms of “deadly embrace.” The first occurs when two users attempt to LOCK the same set of files in separate orders of sequence. The scenario is:

User A locks file A

User B locks file B at the same time

User A attempts to LOCK file B and cannot because User B has it LOCKED

User B attempts to LOCK file A and cannot because User A has it LOCKED

This leaves both users “hung up” attempting to gain control of the files. The solution to this dilemma is the adoption of a simple coding convention: Always LOCK files in the same order (alphabetical works just fine) and trap for other users’ LOCKs. This example demonstrates the principle:

```

LOOP
  LOCK(FileA,1)                !Attempt LOCK for 1 second
  IF ERROR() = 'FILE IS ALREADY LOCKED'
    CYCLE                      ! and try again
  END
  LOCK(FileB,1)                !Attempt LOCK for 1 second
  IF ERROR() = 'FILE IS ALREADY LOCKED'
    UNLOCK(FileA)             !Unlock the locked file
    CYCLE                      ! and try again
  END
  BREAK                        !Break from loop when both locked
END

```

This code will eventually LOCK both files. If FileA is already locked by another user, the loop will try again. The one second pause allows the other user a chance to complete their action. If the first LOCK is successful, the LOCK on FileB is attempted. If FileB is already locked by another user, FileA is immediately unlocked for other user’s use, then re-try sequence occurs again. The BREAK from the LOOP in this example is only allowed after both files are successfully LOCKED.

Mixing the use of HOLD and LOCK can result in the second form of “deadly embrace.” In some file systems, LOCK and HOLD are completely independent, therefore it is possible for one user to HOLD a record in a file, and another user to LOCK that same file. The user with the HOLD cannot write the record back, or even RELEASE it, and the user with the LOCK cannot write to that held record.

This situation may be resolved in one of two ways:

- You may choose to never mix HOLD and LOCK on the same file. This limits you to the use of HOLD only (the most common solution), or LOCK only. This solution must be used in all applications that write to a common set of files.

- You may choose to always trap for held records while the file is LOCKed. This implies that you know how you want to deal with the “deadly embrace” record when it is detected.

The first solution is by far the more commonly used. The second takes you into an area of programming that is probably better served by Transaction Processing, which would include held record trapping.

Summary

- A File must be opened before its records may be accessed.
- The *access mode* determines the type of access DOS grants to the user opening the file and any other users.
- Multi-user programming must always take into consideration the possibility of multiple users accessing the same record at the same time.
- Concurrency checking is done to ensure that user updates don't overwrite other user's changes to the records.
- HOLD is most commonly used in conjunction with concurrency checking to ensure that no other user can change the record while it is being compared for previous changes.
- LOCK is most commonly used to gain exclusive control of a file while you BUILD an INDEX.
- The “deadly embrace” is a programming consideration most easily dealt with through the adoption of consistent program coding conventions.

7 - DEVELOPING CLIENT/SERVER APPLICATIONS

Introduction to Client/Server

Client/Server Defined

What exactly constitutes Client/Server computing? To some, it indicates the use of Groupware (programs such as Lotus Notes), or Object Linking and Embedding (OLE) between OLE client and OLE server applications, or just any program that splits task completion between client workstations and network servers. There are also many “buzzwords” that go along with the subject, such as “downsizing” and “rightsizing.”

At its most basic level, the broadest definition of Client/Server computing comes down to this: *a Client workstation computer sends a message (requesting some kind of service) across a network of some kind to a Server computer, which processes the service request and responds with either a return message (returning requested data) or an acknowledgement.* No matter what form it takes, the ultimate purpose of Client/Server computing is to make the most efficient use of the computing resources available, on an enterprise-wide basis.

Clarion is optimized to create business database applications. This means the most common type of Client/Server applications created with Clarion are the type where the program executes on a Client workstation and accesses a database server engine (frequently SQL-based) located on a database server in the the network.

Therefore, for the purposes of this discussion, the definition of Client/Server computing that we will use is this: *creating database applications that logically partition the workload between the Client workstations and the Server-based database engine so that maximum performance efficiency is maintained throughout the network.*

Types of Client/Server Database Applications

There are two major categories of Client/Server database applications: On Line Transaction Processing (OLTP), and Decision Support Systems (DSS).

On Line Transaction Processing (OLTP)

OLTP applications are built to continually maintain and update the database in real time. This means OLTP applications require high performance from

both the hardware and software so that users are not kept waiting for any requested information for more than a very few seconds.

Decision Support Systems (DSS)

DSS applications are usually used by a company's management for stored or ad hoc queries to monitor the state of the company. This usually means the data does not need to be quite so "up to the minute" and response times are not nearly as critical. Frequently, DSS applications are built to access on-line archival data or a replication of the database that is updated only at timed intervals.

Clarion Client/Server Applications

Typically, Clarion-created Client/Server database applications fall into the On Line Transaction Processing (OLTP) category, rather than the Decision Support Systems (DSS) used for ad hoc queries. Your users can use Clarion's *ReportWriter for Windows* product to generate standard or ad hoc reports from the database (thus providing them DSS functionality).

SQL Database Engines

Most Client/Server applications are "front-ends" running on a Client workstation and requesting data services from a database engine (such as Oracle or Sybase) running on a network Server. Most of these database engines are accessed through Structured Query Language (SQL).

SQL was originally developed as an ad hoc query language to allow users to glean information from a database. Over the years, the SQL language has been enhanced to allow programmatic access to databases in addition to its original ad hoc query mission.

SQL has become the standard tool used to access the database engines used in Client/Server computing—programmatic SQL is the tool most frequently used to perform the data access chores in the programs that execute on Client workstations. However, SQL is not a full-featured programming language (for instance, it has no user interface design capabilities), so the SQL must be "embedded" into a more general purpose programming language (such as C++ or Clarion) to create a complete program.

Clarion and SQL

Clarion's file driver technology allows you to write applications that access SQL databases without the necessity of writing any SQL at all—the Clarion file driver for any SQL database communicates with the back-end database server by automatically generating the SQL statements that the database

engine requires from your standard Clarion language file I/O syntax. Since the Clarion SQL file drivers are specifically designed to “talk to” a single SQL back-end database (except the ODBC driver, of course, which is a generic SQL file driver), the SQL it generates is optimized to use features specific to that back-end database. This means that a Clarion programmer is not *required* to learn (or use) SQL at all to write efficient Client/Server applications.

The fact that a Clarion programmer is not *required* to write SQL does not in any way limit the capability of Clarion to generate Client/Server programs. You can embed your own SQL statement to extend what the file driver does for you, or you can take full control of the database yourself to accomplish any task that needs doing.

There are several ways to directly embed your own SQL statements into a Clarion program, if you so choose. This gives you, the Clarion programmer, the flexibility to just “let it happen” in a standard fashion (by letting Clarion’s file driver handle all the SQL), or to “make it happen” any other way you choose by directly writing your own SQL, as appropriate to the requirements of your individual application.

PROP:SQL is a property of a FILE or VIEW structure that allows you to send any SQL statement directly to the back-end database server.

PROP:SQLFilter is a property of a VIEW structure that allows you to append your own SQL WHERE clause to the generated SQL sent to the back-end database. Beyond these, there are also several other properties available that are used to enhance the SQL generated by the file driver (some are file-driver-specific). In either case, the file driver places the returned data set from the back-end database in the appropriate FILE record buffer, allowing you to use standard Clarion syntax to manipulate its value, or assign the data values to other fields.

Database Design and Network Traffic

The most important aspect to creating efficient Client/Server applications is the design of the database. When properly designed, the database more readily allows you to write your applications such that network traffic can be minimized. When network traffic is kept to a minimum, data response times (the most critical aspect of OLTP systems) can be kept as fast as possible, even when the system is scaled up to the enterprise level.

Referential Integrity Handling

Maintaining the Referential Integrity (RI) of a database is a key element to Relational Database design. Referential Integrity means that, for every One-to-Many (Parent-Child) relationship between tables in the database there exists a Parent record for every Child record (no “orphan” records). To put it in more formal terms, there must be a valid Primary Key value for every existing Foreign Key in the database.

“Orphan” records can occur when the Parent record is deleted, or the Primary Key value (which provides the link to the Foreign Key in the Child record) is changed. Preventing these “orphan” records requires that the database contain rules stating what action will occur when the end-user attempts to delete the Parent record, or change the Primary Key value.

The most common RI rules are “restrict” (do not allow the delete or change) and “cascade” (delete the related Child records or change their Foreign Key values to match the new Primary Key value in the Parent record). A rarely used rule is “clear” (change the Foreign Key values to NULL when the Parent record is deleted or the Primary Key value in the Parent record changes).

RI constraint enforcement is best handled in Client/Server applications by specifying the RI rules on the back-end database server, usually by defining Triggers, Stored Procedures, or Declarative RI statements. By doing this, the database server can automatically handle RI enforcement without sending any Child records across the network to the Client application for processing. For example, if the rule for Delete is “cascade,” the database server can simply perform all the required Child record deletions when deleting the Parent record from the database—without sending anything back across the network to the Client application.

In the Clarion Dictionary Editor, when you establish a relationship between two files (tables), you can also specify the RI rules for that relationship. Since the database server will actually be handling the RI functionality, the most appropriate way to specify the RI rules in the Clarion Data Dictionary would be to specify “no action” so the Client does nothing.

Data Validation

Data validation means the enforcement of business rules (that you specify) as to what values are valid for any particular field in the database. Typical data validation rules enforce such things as: a field may not be left empty (blank or zero), or the field's value must be either one or zero (true or false) or within a certain specified numeric range, or the field's value must exist as a Primary Key value in another file (table).

These data validation rules can be specified either in the Client application or in the back-end database server. The best way to handle implementing data validation rules in your Client/Server applications, so as to generate minimal network traffic, is to specify the business rules in both the Client application *and* the database server:

- By enforcing data validation rules in the Client application you ensure that all data sent to the back-end is already valid. By always receiving valid data the database server will not generate error messages back to the Client application. The net effect of this is to reduce the network traffic back from the database server.
- By enforcing data validation rules on the back-end database server you ensure that the data is always valid, no matter what application is used to update the database—even updating the data with interactive SQL cannot corrupt the data. Therefore, you are covered from both directions.

Enforcing these rules in both your Clarion applications and the database server may seem like a lot of work. However, the Clarion Data Dictionary Editor allows you to specify the most common rules by simply selecting a radio button on the Validity Checks tab of the affected field's definition. By doing this, the actual code to perform the data validation is written for you by the Application Generator's Templates.

Clarion Language Client/Server Support

The Clarion language contains a number of statements that are explicitly designed to support Client/Server application programming.

The VIEW Structure

The VIEW structure is a data structure that automatically defines two standard relational database operations: the PROJECT and JOIN operations. A VIEW will also automatically FILTER and ORDER the result set. The back-end database server performs all these operations, returning only the result set that the Client application needs to perform its work.

Although it is actually possible to create a VIEW structure that JOINS tables from different back-end database servers (such as Oracle and AS/400), this would not be a very efficient way to write a Client/Server application because the JOIN (and any filtering) would have to be processed on the Client, eliminating the advantage of Client/Server computing. Therefore, we will not address this possibility in this article.

PROJECT

A relational PROJECT operation tells the back-end database server to only return to the Client application a specific sub-set of the columns in a table (thereby reducing network traffic). For example, the following VIEW structure will return only the specified fields (columns) from the Students file (table):

```
MyView VIEW(Students)
      PROJECT(STU:LastName,STU:FirstName,STU:Major)
      END
```

JOIN

The relational JOIN operation automatically joins together related rows from multiple tables into a single result set which the database server returns to the Client application. The VIEW structure defaults to a *left outer join* unless the INNER attribute is specified on the JOIN structure. A *left outer join* returns all of the outer rows, whether there are related inner rows to return or not. For those outer records without related inner records, the fields in the inner record are left empty (blank or zero). For example, the following VIEW structure will return all records (rows) in the Students file (table), whether or not there are related Majors file records:

```
MyView VIEW(Students)
      PROJECT(STU:LastName,STU:FirstName,STU:Major)
      JOIN(MAJ:KeyNumber,STU:Major)
      PROJECT(MAJ:Description,MAJ:Number)
      END
      END
```

Adding the **INNER** attribute to the **JOIN** structure specifies an *inner join* which returns only those outer rows with related inner rows. For example, the following **VIEW** structure will return only the records (rows) in the Students file (table), where there are related Majors file records:

```
MyView VIEW(Students)
        PROJECT(STU:LastName,STU:FirstName,STU:Major)
        JOIN(MAJ:KeyNumber,STU:Major),INNER
        PROJECT(MAJ:Description,MAJ:Number)
END
END
```

FILTER

The **FILTER** attribute on a **VIEW** allows you to specify a conditional expression to filter out unwanted records. This will generate a **WHERE** clause in the generated **SQL SELECT** statement. For example, the following **VIEW** will return only those Students file records whose last name is “Taylor”:

```
MyView VIEW(Students),FILTER('STU:LastName = 'Taylor''')
        PROJECT(STU:LastName,STU:FirstName,STU:Major)
END
```

ORDER

The **ORDER** attribute on a **VIEW** allows you to specify the sort order of the result set returned by the database server. This will generate an **ORDER BY** clause in the generated **SQL SELECT** statement. For example, the following **VIEW** returns the Students file records sorted in descending last name and ascending first name order:

```
MyView VIEW(Students),ORDER('-STU:LastName,+STU:FirstName')
        PROJECT(STU:LastName,STU:FirstName,STU:Major)
END
```

Naturally, all these attributes can be combined so that the result set returns to the Client application filtered, ordered, projected and joined. By allowing the back-end database server to do this work, the only network traffic generated is the minimum necessary to give the Client application the requested data.

The BUFFER Statement

The Clarion **BUFFER** statement can have a tremendous impact on Client/Server application performance. **BUFFER** tells the file driver to set up a buffer to hold previously read records and a read-ahead buffer for anticipated record fetches. It also specifies a time period during which the buffered data is considered to be valid (after which the data is re-read from the back-end database server).

When the file driver knows it has buffers to hold multiple records it can optimize the **SQL** statements it generates to the back-end database server. This allows the back-end database server to return a set of records instead of

a single record at a time (also called “fat fetches”). The net effect of this is to change the pattern of network traffic from many small pieces of data to fewer but larger chunks of data, making for more efficient overall network utilization. The most common use of BUFFER would probably be in procedures which allow the end-user to browse through the database.

By setting up buffers to hold already read records, the Client machine fetches records from the local buffer when the user has paged ahead then returns to a previous page of records, instead of generating another request to the back-end database server for the same page of records. This eliminates the network traffic that would normally be generated for subsequent requests for the same set of records.

Setting up read-ahead buffers enables the Client application to anticipate the user's request for the next page of records and receive them from the back-end database server while the user is still examining the first page. Therefore, when the user finally does request the next page, those records are also fetched from the local buffer on the Client machine, giving the end-user apparently instantaneous database retrieval.

For example, the following BUFFER statement tells the file driver to consider 10 records as a single “page,” to buffer 5 previously read pages, to read 2 pages ahead, and to consider the buffered records valid for 5 minutes (300 seconds):

```
BUFFER(MyView,10,5,2,300) !10 recs per page, 5 pages behind, 2 read-ahead
! with a 5 minute timeout
```

Embedded SQL in Clarion

Clarion's file driver technology lets the Clarion programmer learn and use just the Clarion language's file Input/Output syntax, no matter which file system contains the data. The file driver automatically converts the Clarion language statements into whatever form the file system requires for its fulfillment of data requests. Therefore, for file drivers which interface to SQL-based database servers, the file driver itself generates the necessary SQL statements to retrieve the data from the database server.

Given all the information that the file driver can obtain from FILE structure declarations, VIEW structure declarations, and the BUFFER statement, the resulting SQL generated by the file driver can be quite efficient. However, there are still times when an experienced and knowledgeable SQL programmer will want to extend the functionality of the file driver, or take full control to accomplish a task that is best accomplished in SQL—some task that the file driver would not normally generate (such as deleting an entire block of records at once, or updating a single field in all rows of a table to some new value). To cover these circumstances, the Clarion language provides a mechanism to allow sending any SQL statement directly to the back-end database server: PROP:SQL.

PROP:SQL

The syntax of PROP:SQL is the same as any other Clarion language property assignment statement, with the target being the label of any FILE (or VIEW) declaration that uses the SQL file driver. For example:

```
MyFile{PROP:SQL} = 'SELECT * FROM SOMETABLE'
```

This statement sends the specified SQL SELECT statement to the back-end database server. Note that the target of PROP:SQL is *MyFile* while the SQL SELECT statement is directed to *SOMETABLE*. You can send any SQL statement to PROP:SQL, regardless of whether the target of PROP:SQL is affected by the SQL or not—the target file is just the mechanism whereby the correct file driver processes your SQL. Since this SQL SELECT statement will return a result set, you must use the Clarion NEXT statement to retrieve the result set one record at a time, and the FILE declaration referenced in the NEXT statement must have the same number of fields as *SOMETABLE*. Obviously, this specific example does not demonstrate the preferred method of working in Clarion, since the file driver itself will more than adequately generate such simple SQL SELECT statements for you.

The real usefulness of PROP:SQL is to perform SQL functions that have no direct corollary in the Clarion language. One prime example of this is table (file) creation on the database server. The Clarion language CREATE statement will certainly allow you to create new tables in the database and the file driver will generate an appropriate SQL CREATE statement to perform the task. However, the Clarion CREATE statement is limited to the information stored in your FILE declaration. It is much better to use PROP:SQL to send an SQL CREATE statement to the database server, because the SQL CREATE can then implement any data validation constraints or triggers that the database server supports, allowing you to make full use of the capabilities of the database server.

For example, the following code creates a Students table with First and Last Names, Major, and Dormitory assignment columns:

```
MyView{PROP:SQL} = 'CREATE TABLE      STUDENTS'           & |
                    '(IDNUMBER  INTEGER NOT NULL, '       & |
                    'LASTNAME   VARCHAR(25) NOT NULL, '   & |
                    'FIRSTNAME  VARCHAR(25) NOT NULL, '   & |
                    'MAJOR      VARCHAR(10) NOT NULL, '   & |
                    'DORMITORY  INTEGER, '                & |
                    'PRIMARY KEY (IDNUMBER), '           & |
                    'FOREIGN KEY MAJORS_IN (MAJOR), '     & |
                    'REFERENCES MAJORS, '                & |
                    'ON DELETE  RESTRICT)'
```

This SQL CREATE statement specifies two items that are not possible to specify using the Clarion CREATE statement. The NOT NULL attribute specifies that data must be present in these columns whenever a new record is added to the database. The ON DELETE clause specifies that the RESTRICT Referential Integrity constraint applies when deleting a Majors

record, so that no Student record can have a Major field value that doesn't exist in the Majors table.

Another primary use of PROP:SQL is to call any stored procedures that you have created in your back-end database. In most SQL-based database servers, a stored procedure is a pre-compiled set of SQL statements that allow you to pre-define actions for the database engine to take when the procedure is called. Since stored procedures are a part of the database, they can be used by any application that accesses the database, which helps enforce consistent interaction with the database across all applications.

PROP:SQLFilter

PROP:SQLFilter is very similar to PROP:SQL. PROP:SQLFilter allows you to specify a filter condition for a VIEW structure using SQL syntax instead of the Clarion syntax required in the FILTER attribute.

By default, the PROP:SQLFilter filter condition overrides any expression in the FILTER attribute. However, by beginning your PROP:SQLFilter expression with a plus sign (+), the file driver will append your PROP:SQLFilter statement to the FILTER attribute's generated SQL.

The advantage of using PROP:SQLFilter instead of the FILTER attribute is to allow the WHERE clause to use the database-specific or SQL-specific capabilities that you can include by using your own SQL (such as an SQL IN clause). For example, the following code uses a FILTER condition to limit the VIEW to only those students with "Computer Science" as their major:

```
MyView VIEW(Students),FILTER('STU:Major = 'Computer Science''')
      PROJECT(STU:LastName,STU:FirstName,STU:Major)
      END
```

The following code replaces the FILTER with PROP:SQLFilter:

```
MyView VIEW(Students)
      PROJECT(STU:LastName,STU:FirstName,STU:Major)
      END
CODE
OPEN(MyView)
MyView{PROP:SQLFilter} = 'Students.Major = 'Computer Science'''
```

Both of these examples return the same result set to your Clarion application.

The following code uses the FILTER condition to limit the VIEW to only those students with "Computer Science" as their major, then appends an SQL IN clause to the generated WHERE clause to further limit the result set to only those Computer Science students who also live in dormitories 1, 5, or 9:

```
MyView VIEW(Students),FILTER('STU:Major = 'Computer Science''')
      PROJECT(STU:LastName,STU:FirstName,STU:Major,STU:Dormitory)
      END
CODE
OPEN(MyView)
MyView{PROP:SQLFilter} = '+Students.Dormitory IN (1, 5, 9)'
```

NULL Data Handling

One concept common in SQL-based database servers is the concept of NULL data. The concept of a null “value” in a field of a FILE or VIEW indicates that the user has never entered data into the field. Null actually means “value not known” for the field. This is completely different from a blank or zero value, and makes it possible to detect the difference between a field which has never had data, and a field which has a (true) blank or zero value. The Clarion language supports NULL data handling through the NULL, SETNULL, and SETNONULL procedures.

In expressions, null does not equal blank or zero. Therefore, any expression which compares the value of a field from a FILE or VIEW with another value will always evaluate as unknown if the field is null. This is true even if the value of both elements in the expression are unknown (null) values. For example, the conditional expression `Pre:Field1 = Pre:Field2` will evaluate as true only if both fields contain known values. If both fields are null, the result of the expression is also unknown.

```

Known = Known           !Evaluates as True or False
Known = Unknown        !Evaluates as unknown
Unknown = Unknown     !Evaluates as unknown
Unknown <> 10          !Evaluates as unknown
1 + Unknown            !Evaluates as unknown

```

The only four exceptions to this rule are boolean expressions using OR and AND where only one portion of the entire expression is unknown and the other portion of the expression meets the expression criteria:

```

Unknown OR True        !Evaluates as True
True OR Unknown        !Evaluates as True
Unknown AND False     !Evaluates as False
False AND Unknown     !Evaluates as False

```

Support for null “values” in a FILE or VIEW is entirely dependent upon the file driver. Most SQL-based database systems support the null field concept, while most non-SQL databases do not.

You use the Clarion NULL procedure to detect whether a data item returned from the back-end database is null or not. For any fields that should remain null when you re-write the data to the database, you must explicitly call SETNULL just before writing the data back to the database. For example:

```

NEXT(MyTable)           !Get data
MyFieldFlag = NULL(MyField) !Remember whether a field is null or not
!
!Process the data
!
IF MyFieldFlag AND NOT MyField !Detect null field still empty
  SETNULL(MyField)           ! and reset it to null
END
PUT(MyTable)             !Write the data back

```

Error Handling

Whenever any I/O operation executes there is the possibility of an error condition occurring. No matter which back-end database server you use, the Clarion file driver for that database maps the most common errors to the appropriate standard Clarion error codes. However, there are always some errors for which there is no direct Clarion equivalent.

Whenever an error occurs for which there is no direct Clarion equivalent, the Clarion `ERRORCODE` procedure returns 90 and the `ERROR` procedure returns "File Driver Error." To determine the exact error returned by the back-end database server in this case, Clarion provides the `FILEERRORCODE` and `FILEERROR` procedures.

The `FILEERRORCODE` and `FILEERROR` procedures return the back-end database server's error values when the current Clarion `ERRORCODE` is 90. This allows you to detect any error the database server can issue, then look in the back-end database server's documentation for the possible causes and remedies for whatever error has occurred.

PART II



ADVANCED PROGRAMMING RESOURCES

8 - CUSTOMIZING THE DEVELOPMENT ENVIRONMENT

An Overview

The Clarion Development Environment was set up in an optimal manner for developing applications. However, it is also extensible to allow you to modify it to meet your specific needs. You can add menu items to call built-in applets (a small application that performs a single task) or any external tools you wish to use. Each portion of the Clarion Development Environment is an applet that can be called to perform a task. For example, you can call the Source Editor (CWedt) to open and edit a template file.

The configuration settings file —Clarion4.INI—controls this extensibility. This is a standard Windows .INI file that the Clarion GETINI and PUTINI statements can modify.

As is the case with most Windows applications, the Clarion Development Environment reads the .INI file upon opening and writes to it upon closing. Keep this in mind when editing it. If you make any changes to the .INI file while Clarion is open, these changes will be overwritten when you close Clarion. For this reason, you should use Notepad or any other text editor to edit the .INI file.

Note: We strongly recommend making a backup copy before modifying your .INI file.

Command Line Parameters

There are several command line parameters you can use when you call C5EE.EXE to invoke specified initial actions. All of them will automatically disable the Pick list and prevent it from appearing as the opening dialog.

- Pname** Sets the initial Project to the passed *name*.
- Mname** Makes the Project *name*.
- Rname** Makes and runs the Project *name*.
- Dname** Makes and debugs the Project *name*.
- Ename** Opens the *name* file.

Example:

```
C5EE.EXE -Rtutorial.app  
C5EE.EXE -Ehello.clw
```

Non-Modifiable Clarion4.INI File Sections

User Information

The Clarion Development Environment stores your user information (Name, Company, Serial Number, and Key Number) in the **[user]** section of the .INI file. You should not modify this section.

Paths

The last opened project or Application is stored in the **[paths]** section of the .INI file. This is updated every time the environment is closed.

You can modify this section of the INI file, but it is always overwritten when Clarion closes.

Example:

```
[paths]
prjfile=C:\CW20\APPS\MYAPP.APP
```

Environment Options

The settings specified in the **[Environment]** section control the appearance of the Clarion development environment. Two settings are specified in the **[Environment]** section which are always overwritten when Clarion closes:

maximized=on|off

Specifies whether the environment opens maximized.

quickstart=on|off

Specifies whether Quick Start is on by default.

Other settings appear in this section which may be modified:

windows95dlgs=on|off

Specifies whether to use Windows 95-style common dialogs.

wallpaper=*bmpfilename*

Specifies the .BMP file the environment displays for wallpaper.

wallpapermode=*tiled|centered|*_{full}

Specifies how to display the .wallpaper BMP file: tiled, centered, or stretched to fill the environment client area.

autopick=on|off

Specifies whether the Pick dialog appears when the environment first opens.

Example:

```
[Environment]
maximized=off
quickstart=off
windows95dlg=off
wallpaper=c:\clarion5\images\c5wall.bmp
wallpapermode=tiled
autopick=on
```

The Clarion Applets

The Clarion Development Environment has several built-in applets that you can call from a User Defined Menu or a User Defined Tab on one of the File Dialogs (New, Open, or Pick). These applets are “registered” through the **[Clarion Applets]** section of the INI file.

You should not modify this section. The applets are documented here so you know which ones you can call from menus or tabs.

Applets marked with an asterisk (*) are for internal use only and cannot be called separately. They are listed here for informational purposes only.

C5edt	The Source Code Editor
C5cif	The Compiler Interface
C5scr	The Window Formatter*
C5prj	The Project Editor
C5rpt	The Report Formatter*
C5adm	The Data Dictionary Editor
C5gen	The Application Generator
C5asv	Application Generator Services*
C5frm	The Formula Editor*
C5brw	The Database Manager
C5qck	The QuickStart Wizard*

These are listed in the **[Clarion Applets]** section of the .INI file.

For example:

```
[Clarion Applets]
_version=8
CWedt16=on
CWcif16=on
CWscr16=on
CWprj16=on
CWrpt16=on
CWadm16=on
CWgen16=on
CWasv16=on
CWfrm16=on
CWbrw16=on
CWqck16=on
```

Modifying the Clarion Environment

Specifying User Defined Applications

To call an application or applet from the Clarion development environment, you must first define the application in the **[User Applications]** section of CLARION4.INI. This “registers” the application with the environment. To add your own, modify the **[User Applications]** section of CLARION4.INI in the following manner:

AppReference=CommandLine

AppReference

The name by which the application is referenced.

CommandLine

The text of the command to be launched. This text may contain either the %f or %a expansion macros.

Example:

```
[User Applications]
notepad=notepad %f
wordpad=wordpad %f
notepad.2=notepad c:\windows\win.ini
notepad.3=notepad %a.txt
```

Note: If your application is in a directory that is not in your PATH and is using Long Filenames (Windows 95) you must use the DOS equivalent of the directory and filenames. For example, if Wordpad is in the C:\Program Files\Accessories directory, you must refer to it as: C:\Progra~1\ACCESS~1\wordpad.

Notice the last two lines of the example reference notepad.2 and notepad.3. This defines notepad.2 to open Notepad to edit the WIN.INI file. Notepad.3 is defined to open (or create) a file using the name of the current application with a .TXT extension. You can add an extension User Applications to define multiple occurrences of the same application to behave in different fashions.

The %f expansion macro shown in the example denotes the current filename. This is passed as a command line parameter to the application. Valid expansion macros you can use as command line parameters are:

%f	Filename and Path
%-f	Filename only
%a	Current Application (*.APP) or Project (*.PRJ)

Additionally, you can add an extension to an expansion macro to change the extension of the filename passed to it. For example, calling notepad with the parameter %a.txt would open a file using the current application's filename

and a .TXT extension. This enables you to define a User Application that you can call to open (or create) a text file matching the current application name.

Adding choices to the Clarion Menu

To call an external application from the Clarion development environment, you must first define the application in the [User Applications] section of CLARION4.INI as described above. This “registers” the application with the environment.

Next, you add Menu(s) and define the menu selections in the [User Menus] section in the following manner:

n=MenuDisplayText/MenuDisplayText/Applet

n The number of the entry

MenuDisplayText

The text to display in the action bar.

ItemDisplayText

The text to display in the Drop-Down List.

Applet

The applet to call.

Example:

```
[User Menus]
_version=1
1=&Editors/&Wordpad|wordpad
2=&Editors/&Notepad|notepad
3=&Utilities/&Calculator|Calculator
4=&Utilities/&Paint|Paint
```

Note: The first line (`_version=1`) denotes a version number used internally. You should not modify that line.

Adding choices to the Clarion Setup Menu

You can also add choices to the Setup Menu in the Clarion Development Environment. To call an external application from the Clarion development environment, you must first define the application in the **[User Applications]** section of CLARION4.INI as described above. This “registers” the application with the environment.

Next, you add menu choices and define the menu selections in the **[Setup Menu]** section as shown in numbers 10 and 11 below. Number 11 uses a User Defined Application (See *Specifying User Defined Applications*). Numbers 4, 8, and 10 create separators by using the *n=-* syntax.

n=Text|Application

n The number of the entry

Text

The text to display on the menu.

Application

The Clarion applet or application to open. Any external applications must be defined in the **[User Applications]** section.

Example:

```
[Setup Menu]
_version=10
1=&Editor Options|CWedt
2=&Dictionary Options|CWadm
3=&Application Options|CWgen.1
4=-
5=&Template Registry|CWgen.103
6=Data&base Driver Registry|CWas1.38
7=&VBX Custom Control Registry|CWas1.37
8=-
9=Edit Redirection &File|CWas1.36
10=-
11=Edit WIN.INI|notepad.2
```

Note: The first line (*_version=10*) denotes a version number used internally. You should not modify that line.

Adding File Masks to File Types Drop Down lists

The **[File Types]** section defines the masks used by the drop-down list in any of the Clarion file dialogs (New, Open, or Pick) and the applet or application which opens or creates the file. One additional entry should follow, in the format of **ALL=*n***. This defines which should be used for ALL files, allowing more than one entry to use *.* as the file mask.

Note: The first line (_version=8**) denotes a version number used internally. You should not modify that line.**

The defaults are :

```
[File Types]
_version=8
1=Application (*.app)=*.app|CWgen
2=Clarion source (*.clw)=*.clw|CWedt
3=Dictionary (*.dct)=*.dct|CWadm
4=Project (*.prj)=*.prj|CWprj
5=Database files=*.tps;*.dbf;*.dat;*.db|CWbrw
6=All Database files (*.*)=*..*|CWbrw
7=All files (*.*)=*..*|CWedt
all=7
```

To add your own, modify the **[File Types]** section of CLARION4.INI in the following manner:

n=DisplayText=Filemask|Application

n The number of the entry

DisplayText The text to display in the Drop-Down List.

Filemask The mask to use when listing files.

Application The Clarion applet or external application to use to open the file. External applications must be defined in the **[User Applications]** section.

Example:

```
[File Types]
1=Application (*.app)=*.app|CWgen
2=Clarion source (*.clw)=*.clw|CWedt
3=Dictionary (*.dct)=*.dct|CWadm
4=Project (*.prj)=*.prj|CWprj
5=Text (*.txt)=*.txt|notepad
6=Doc (*.doc)=*.doc|wordpad
7=Database files=*.tps;*.dbf;*.dat;*.db|CWbrw
8=All Database files (*.*)=*..*|CWbrw
9=All files (*.*)=*..*|CWedt
all=9
```

Adding Tabs to New, Open, or Pick File Dialogs

Each of the Clarion File dialogs can be modified to include TABs of your choice.

To add your own, modify the **[Pick Dialog]**, **[New Dialog]**, or **[Open Dialog]** sections of CLARION4.INI in the following manner:

n=TabText=FileMask|Application

n The number of the entry

TabText

The text to display on the Tab.

FileMask

The extension of the files displayed.

Application

The Clarion applet or application to use to open the file. Any external applications must be defined in the **[User Applications]** section.

Note: The first line (**_version=8**) denotes a version number used internally. You should not modify that line.

Example:

```
[Pick Dialog]
_version=8
1=&Application=app|CWgen
2=&Dictionary=dct|CWadm
3=&Project=prj|CWprj
4=Data&base=*|CWbrw
5=&Source=clw|CWedt
6=&Text=txt|notepad
7=&Doc=doc|wordpad
8=A&ll=*|CWas1
lines=2
default=-8
```

```
[Open Dialog]
_version=8
1=&Application=app|CWgen
2=&Dictionary=dct|CWadm
3=&Project=prj|CWprj
4=Data&base=*|CWbrw
5=&Source=clw|CWedt
6=A&ll=*|CWas1
7=&Text=txt|notepad
8=&Doc=doc|wordpad
lines=2
default=-2
```

```
[New Dialog]
_version=8
```

```
1=&Application=app|CWgen  
2=&Dictionary=dct|CWadm  
3=&Project=prj|CWprj  
4=&Source=clw|CWedt  
5=&Text=txt|notepad  
6=&Doc=doc|wordpad  
lines=2  
default=-1
```

The last two lines of each of these sections affect the TABs on each of the File Dialogs. The `lines=2` specifies that enough space should be allotted for two rows of TABs. This does not cause TABs to split onto two lines, that is controlled automatically by the space needed.

The `default=` line controls which TAB appears on top when the dialog is called. If the number is positive, the TAB equated to the number will always open on top. If the number is negative, the environment will save the last open tab's number to that position causing the dialog to reopen with the TAB which was selected last on top.

Specifying Make File Types

The Compiler Interface uses Project files or Application files when compiling and linking. If you like, you can define alternate extensions for these types of files. By defining your own, for example, you can compile a project file with an extension of .MAK. The file must be in the same format as other project files (see *Chapter 13—The TopSpeed Project System*).

To add your own, modify the **[Make File Types]** section of CLARION4.INI in the following manner:

n=DisplayText=Filemask|Applet

n The number of the entry

DisplayText

The text to display in the Drop-Down List.

Filemask

The mask to use when listing files.

Applet

The Clarion applet to use to open the file.

Note: The first line (***_version=8***) denotes a version number used internally. You should not modify that line.

Example:

```
[Make File Types]
_version=8
1=Application (*.app)=*.app|CWgen
2=Project File (*.prj)=*.prj|CWcif
3=Text Project File (*.pr)=*.pr|CWcif
4=Make File (*.mak)=*.mak|CWcif
```

Print Specifications

By default, all files printed from the environment have a header listing the filename, date, and page number. This is controlled by the **[printing]** section of the INI file. You can modify this section to print a different header if desired. This uses any of the following expansion macros:

%F	Filename
%D	Current System Date
%T	Current System Time
%P	Current Page Number

Note: The first line (***_version=1***) denotes a version number used internally. You should not modify that line.

```
[printing]
_version=1
header=File: %F Date: %D Time: %T Page: %P
```

Environment Option Settings

Auto Populate Options

The [Auto Populate] section specifies where controls are placed when auto populating (by double-clicking) with the Fields Box in the formatters.

startX=5

Specifies the starting position on the X axis (horizontal axis) for the first control placed.

startY=5

Specifies the starting position on the Y axis (vertical axis) for the first control placed.

incrementX=50

Specifies the number of dialog units to increment the horizontal offset for controls from the associated prompt.

incrementY=16

Specifies the number of dialog units to increment the vertical space between auto-populated controls.

alignment=top

Specifies whether to vertically align controls to the top, bottom, or center of its associated prompt.

Example:

```
[Auto Populate]
startX=5
startY=5
incrementX=50
incrementY=16
alignment=top
```

Dictionary Options

Data Dictionary Options are controlled by the **[Administrator]** section of the INI file. These options are all modifiable through the development environment, but you can also change them in the INI file.

Default driver=DriverName

The default database driver for new files in a dictionary

Set threaded=on|off

Specifies whether or not new file definitions default to adding the THREAD attribute.

Assign message=on|off

Specifies whether to use the field descriptions you specify when defining a field as the text for the MSG attribute.

Order files=on|off Specifies whether to display files in alphabetical order or in the order in which they were created.

Display field type=on|off

Specifies whether to display the field's data type in the Fields list.

Display field prefix=on|off

Specifies whether to display the prefix in the Fields list.

Display field picture=on|off

Specifies whether to display the screen display picture in the Fields list.

Display field description=on|off

Specifies whether to display the description in the Fields list.

Display key type=on|off

Specifies whether to display the key type in the Key list.

Display key prefix=on|off

Specifies whether to display the prefix in the Key list.

Display key description=on|off

Specifies whether to display the description in the Key list.

Display key primary=on|off

Specifies whether to display "Primary" if the Key is the Primary Key.

Display key unique=on|off

Specifies whether to display "Unique" if the Key is unique.

Display key attributes=on|off

Specifies whether to display the other key attributes in the Key list.

Display file driver=on|off

Specifies whether to display the file driver in the Files list.

Display file prefix=on|off

Specifies whether to display the file prefix in the Files list.

Display file description=on|off

Specifies whether to display the file description in the Files list.

Example:

```
[Administrator]
Default driver=TOPSPEED
Set threaded=on
Assign message=on
Order files=off
Display field type=on
Display field prefix=off
Display field picture=on
Display field description=on
Display key type=off
Display key prefix=off
Display key description=on
Display key primary=off
Display key unique=off
Display key attributes=off
Display file driver=off
Display file prefix=off
Display file description=on
```

Project System Options

Project System Options are controlled by the **[Project System]** section of the INI file. These options are all modifiable through the development environment, but you can also change them in the INI file.

automake=on|off

Specifies whether an application should be automatically compiled and linked (if necessary) before running.

autosave=on|off

Specifies whether an application should be automatically saved before it is compiled and linked.

runmin=on|off

Specifies whether the development environment should be minimized during execution of the application.

runwait=on|off

Specifies whether the Project System should suspend Clarion until after you terminate the application upon executing it with the Run command.

newtype=<extension>

Specifies any new project file types you have defined in the [Make File Types] section of the .INI file. This line is added automatically by the environment when a new Make File Type is added.

Default32bit=on|off

Specifies any new project is created as 32-bit by default.

Debugresume=on|off

Specifies whether the debugger starts in “resume previous debug session” mode.

Example:

```
[Project System]
automake=on
autosave=on
runmin=off
runwait=off
newtype=MAK
```

Application Generator Options

Application Generator Options are controlled by the **[Application]** section of the INI file. These options are all modifiable through the development environment, but you can also change them in the INI file.

CondGeneration=on|off

Specify ON to ensure only source code modules changed since the last make should be compiled.

DebugGeneration=on|off

Specifies whether or not to write to a text file to log events for the Application Generator, and turns logging on and off. In case of a fatal error by the Application Generator, this log provides a trace to identify where the problem occurred. You specify the file name in the Debug Filename entry. This is particularly useful when designing templates.

Repeat Procedures=on|off

Specifies whether or not the Application Generator displays the names of all procedures, as it encounters them in the source code modules, in the progress box displayed during code generation.

PopulateMain=on|off

Specifies whether or not the Application Generator writes procedures to the main source code module. When this option is off, the main module only contains the MAIN procedure, program global code, internally generated procedures standard for every application. All other procedures reside in other file(s).

RequireDictionary=on|off

Specifies whether each new application must have a data dictionary.

DefaultDictionary=dictionaryname

Specifies the name of the default data dictionary.

MultiUser=on|off

Specifies whether to use file management options for multiple developer projects. See the *Multi-Programmer Development* appendix in the *User's Guide* for more information.

ApplicationWizard=on|off

Specifies whether to use the Application Wizard by default when creating a new application. You can override this choice when creating an application by checking or unchecking the Application Wizard box in the Application Properties dialog.

ProcedureWizard=on|off

Specifies whether to use a the appropriate Procedure Wizard by default when creating a new procedure. You can override this choice when creating a procedure by checking or unchecking the Procedure Wizard box in the Select Procedure Type dialog.

NameClash=n

Specifies how the Application Generator handles procedure names from an imported application file which clash with procedure names already resident.

The choices are:

- 0 - Query on First Clash
- 1 - Ask for Alternative
- 2 - Auto Rename
- 3 - Replace Previous

ModuleProcs=n

Specifies the number of procedures that the Application Generator writes to each source code module. This can affect compile time when used with Conditional Generation turned on. Specifying one procedure per module, for example, means that each successive compile rebuilds only those procedures changed since the last one, and no more. The down side to this is that it requires more disk space. Generally, a smaller number is faster.

Debug File=filename

Specifies the file to which template debug information is written.

DisableField=on|off

Specifies whether or not template-generated field-specific (#FIELD) prompts will not display. This does not disable prompts created by Control Templates.

Cw21ProcedureCall=on|off

When on, specifies the Procedures button on a Procedure Properties dialog calls the Clarion version 2 Procedures dilaog.

LocalMap=on|off

When on, specifies local MAP structures are generated in each source module.

TranslateControl=on|off

When on, specifies a dialog appears when controls are populated asking whether to populate the control itself or a related Control Template.

AskOnOK=on|off

When on, specifies an action confirmation dialog appears when exiting with the OK button.

AskOnCancel=on|off

When on, specifies an action confirmation dialog appears when exiting with the Cancel button.

AskOnClose=on|off

When on, specifies an action confirmation dialog appears when exiting.

LegacyAction=0|1|2

Specifies the appearance of Legacy embeds in the Embeditor and Embeds dialog: 0 = Show All and Generate All, 1 = Show Filled and Generate All, 2 = Ignore All.

ShowPriority=on|off

When on, specifies priority embed numbers appear as comment in the Embeditor.

AlphaSortEmbeds=on|off

When on, specifies embed points appear alphabetically in the Embeds dialog.

CommentEmbeds=on|off

When on, specifies embed points appear with comments in the generated source and the Embeditor.

Example:

```
[Application]
CondGeneration=on
DebugGeneration=off
Repeat Procedures=on
PopulateMain=off
RequireDictionary=off
MultiUser=off
ApplicationWizard=on
ProcedureWizard=on
NameClash=0
ModuleProcs=1
Debug File=c:\cw20\apps\mydebug.txt
DisableField=off
```

Dictionary Synchronization Options

Dictionary Synchronization Options are controlled by the [**Control Synchronization**] section of the INI file. These options are all modifiable through the development environment, but you can also change them in the INI file.

ApplicationLoad=on|off

Specify ON to re-synchronize the application with the dictionary every time the application is opened.

Windows=on|off

Specify ON to re-synchronize windows with the dictionary when the application is re-synchronized.

Reports=on|off

Specify ON to re-synchronize reports with the dictionary when the application is re-synchronized.

SyncVariables=on|off

Specify ON to re-synchronize controls for memory variables when the application is re-synchronized.

PrimaryAttrOnly=on|off

Specify ON to re-synchronize only the “primary” attributes.

ChangeControlTypes=on|off

Specify ON to allow control types to change.

SyncDropNonDrop=on|off

Specify ON to allow a LIST with the DROP attribute to change to a LIST without the DROP attribute.

ClearHelpEtc=on|off

Specify ON to clear the Help tab attributes for a control when omitted in the dictionary.

OverrideSize=on|off

Specify ON to allow the dictionary to override the size of controls.

Refreeze=on|off

Specify ON to refreeze a frozen control after synchronization.

IgnoreFreeze=on|off

Specify ON to synchronize all controls, despite their freeze settings.

SyncListbox=on|off

Specify ON to synchronize LIST box FORMAT strings.

SyncHeading=0|1|2|3

Specify heading synchronization: 0 for Always, 1 for In Dictionary, 2 for Dictionary and Window, and 3 for Never.

WarnDialog=on|off

Specify ON to display a warning dialog if problem occur during synchronization.

WarnOnResize=on|off

Specify ON to add a warning to the log file when a control changes size.

WarnFile=textfile.txt

Specify the name of the .TXT file to receive the synchronization report.

ClearAlrtEtc=on|off

Specify ON to clear all miscellaneous attributes if blank in the dictionary.

ClearFont=on|off

Specify ON to clear the FONT attribute if blank in the dictionary.

ClearCursor=on|off

Specify ON to clear the CURSOR attribute if blank in the dictionary.

ClearIcon=on|off

Specify ON to clear the ICON attribute if blank in the dictionary.

ClearAlert=on|off

Specify ON to clear the ALRT attribute if blank in the dictionary.

ClearKey=on|off

Specify ON to clear the KEY attribute if blank in the dictionary.

ClearColor=on|off

Specify ON to clear the COLOR attribute if blank in the dictionary.

ClearTally=on|off

Specify ON to clear the TALLY attribute if blank in the dictionary.

Example:

```
[Control Synchronization]
ApplicationLoad=on
Windows=on
Reports=on
SyncVariables=off
MajorAttrOnly=off
ChangeControlTypes=on
SyncDropNonDrop=off
ClearHelpEtc=off
ClearAlrtEtc=off
OverrideSize=on
Refreeze=on
IgnoreFreeze=off
SyncListbox=on
SyncHeading=2
WarnDialog=on
WarnOnResize=off
WarnFile=sync.txt
ClearFont=off
ClearCursor=off
ClearIcon=off
ClearAlert=off
ClearKey=off
ClearColor=off
ClearTally=off
ClearKeep=off
ClearBreak=off
```

Template Registry Options

Template Language code can be logically split among many files. Clarion uses the files to produce one logical template set for creating applications. The Registry Options are mainly for programmers who produce their own template files or make modifications to the default templates. These options are all modifiable through the development environment, but you can also change them in the INI file. Template Registry Options are controlled by the **[Registry]** section of the INI file.

Reregister if changed=on|off

Specify ON to automatically re-register your templates when the Application Generator detects a change.

Update Template Chain=on|off

Specify ON to automatically update the Template files when making a change in the Template Registry.

Recreate deleted templates=on|off

Specify ON to specify the Application Generator should re-generate the .TPL and .TPW files from REGISTRY.TRF, should the files be deleted.

MessageLines=3

Specifies what displays during generation. The choices are:

- 0-No Messages
- 1-Module Names
- 2-Module and Procedure Names
- 3-All Messages

Example:

```
Reregister if changed=on
Update Template Chain=off
Recreate deleted templates=off
MessageLines=3
```

Window Formatter Options

Window Formatter Options are controlled by the [**Window Formatter**] section of the INI file. These options are all modifiable through the development environment, but you can also change them in the INI file.

grid=on|off

Specifies whether or not the Snap to Grid function is enabled on the sample window.

showtoolbox=on|off

Toggles displaying the Controls tool box.

showalignbox=on|off

Toggles displaying the Alignment Box.

showpropertybox=on|off

Toggles displaying the Property Box.

showfieldbox=on|off

Toggles displaying the Fields Box

tbposx=*n*

The horizontal position of the Controls Toolbox.

tbposy=*n*

The vertical position of the Controls Toolbox.

tbposw=*n*

The width of the Controls Toolbox.

tbposh=*n*

The height of the Controls Toolbox.

pbposx=*n*

The horizontal position of the Property Box.

pbposy=*n*

The vertical position of the Property Box.

pbposw=*n*

The width of the Property Box.

pbposh=*n*

The height of the Property Box.

abposx=*n*

The horizontal position of the Alignment Box.

abposy=*n*

The vertical position of the Alignment Box.

abposw=*n*

The width of the Alignment Box.

abposh=*n*

The height of the Alignment Box.

Example:

```
[window formatter]
grid=off
showtoolbox=on
showalignbox=on
showpropertybox=on
showfieldbox=on
tbposx=720
tbposy=286
tbposw=116
tbposh=199
pbposx=279
pbposy=122
pbposw=338
pbposh=65
abposx=835
abposy=126
abposw=72
abposh=187
```

Control Default Size Options

Control Default Size Options are controlled by the [**Control Defaults**] section of the INI file. These options are all modifiable through the development environment under the Window Formatter Options, but you can also change them in the INI file.

All these options receive the default value for the size (width or height) of the specified control. If the value assigned is zero (0), the size defaults to the default values assigned by the runtime library (no value appears in the control's AT attribute). Positive values indicate the default value for the control if no other control's of that type have been populated (otherwise, the size of the most common control of that type is used). Negative values indicate the (absolute) value for the control, whether other control's of that type have been populated or not.

entry_h=*n*

Specifies the default height of an ENTRY control.

entry_w=*n*

Specifies the default width of an ENTRY control.

button_h=*n*

Specifies the default height of a BUTTON control.

button_w=*n*

Specifies the default width of a BUTTON control.

spin_h=*n*

Specifies the default height of a SPIN control.

spin_w=*n*

Specifies the default width of a SPIN control.

text_h=*n*

Specifies the default height of a TEXT control.

text_w=*n*

Specifies the default width of a TEXT control.

list_h=*n*

Specifies the default height of a LIST control.

list_w=*n*

Specifies the default width of a LIST control.

combo_h=*n*

Specifies the default height of a COMBO control.

combo_w=*n*

Specifies the default width of a COMBO control.

Example:

```
[control defaults]
entry_h=-10
entry_w=0
button_h=-14
button_w=48
spin_h=-10
spin_w=0
text_h=50
text_w=50
list_h=100
list_w=0
combo_h=-10
combo_w=0
```

Report Formatter Options

Report Formatter Options are controlled by the **[Report Formatter]** section of the INI file. These options are all modifiable through the development environment, but you can also change them in the INI file.

grid=on|off

Specifies whether or not the Snap to Grid function is enabled on the sample report.

showtoolbox=on|off

Toggles displaying the Controls tool box.

showalignbox=on|off

Toggles displaying the Alignment Box.

showpropertybox=on|off

Toggles displaying the Property Box.

showfieldbox=on|off

Toggles displaying the Fields Box

tbposx=*n*

The horizontal position of the Controls Toolbox.

tbposy=*n*

The vertical position of the Controls Toolbox.

tbposw=*n*

The width of the Controls Toolbox.

tbposh=*n*

The height of the Controls Toolbox.

pbposx=*n*

The horizontal position of the Property Box.

pbposy=*n*

The vertical position of the Property Box.

pbposw=*n*

The width of the Property Box.

pbposh=*n*

The height of the Property Box.

abposx=*n*

The horizontal position of the Alignment Box.

abposy=*n*

The vertical position of the Alignment Box.

abposw=*n*

The width of the Alignment Box.

abposh=*n*

The height of the Alignment Box.

Example:

```
[report formatter]
grid=off
showtoolbox=on
showalignbox=off
showpropertybox=off
showfieldbox=off
tbposx=540
tbposy=26
tbposw=62
tbposh=259
pbposx=113
pbposy=145
pbposw=338
pbposh=65
abposx=540
abposy=76
abposw=88
abposh=229
```

Editor Options

The Source Editor's Options are controlled by the **[editor]** section of the INI file. These options are all modifiable through the development environment, but you can also change them in the INI file.

Note: The first line (**_version=3**) denotes a version number used internally. You should not modify that line.

font=system

Specifies the font to use in the editor. The default is the system font—the default monospaced font on your system.

maximized=off

Specifies whether the Source Editor window is maximized.

insert=on

Specifies whether the Source Editor is in Insert mode.

Example:

```
[editor]
_version=3
font=system
maximized=off
insert=on
```


9 - USING CLARION AS A DDE SERVER

Overview

Dynamic Data Exchange (DDE) is a very powerful Windows tool that allows a user to send or access data from another separately executing Windows application. The Clarion development environment is a DDE server which you can call from a client application. Using Clarion as a DDE server allows you to write programs which call the Clarion development environment to perform a task or a set of tasks.

For example, you can write a program to register a new template class during the template set's installation program. Another example is a program that manages all of your development projects and will generate, compile, and link multiple applications.

DDE is based upon establishing "conversations" (links) between two concurrently executing Windows applications. One of the applications acts as the DDE server to provide the data, and the other is the DDE client that receives the data. A single application may be both a DDE client and server, getting data from other applications and providing data to other applications. Multiple DDE "conversations" can occur concurrently between any given DDE server and multiple clients.

When Clarion opens, it establishes itself as a DDE server. To accomplish this it must:

- Register with Windows as a DDE server, using the `DDESERVER` procedure.
- Provide services which are available to client applications. The available services are detailed in this chapter.
- When DDE is no longer required, it terminates the link by using the `DDECLOSE` statement. It will terminate the DDE connection automatically when the user closes Clarion.

To be a DDE client, a Clarion application must:

- Open at least one window, since all DDE services must be associated with a window. Additionally, the application should set the `SYSTEM's DDETimeout` property (`SYSTEM{Prop:DDETimeout} = nn`) to an appropriate amount of time (depending on the service the application will request).
- Open a link to a DDE server as its client, using the `DDECLIENT` procedure.

The DDE process posts DDE-specific field-independent events to the ACCEPT loop of the window that opened the link between applications as a server or client.

- Ask the server for data, using the DDEREAD statement, or ask the server for a service using the DDEEXECUTE statement.
- When DDE is no longer required, terminate the link by using the DDECLOSE statement.

The DDE procedures are prototyped in the DDE.CLW file, which you must INCLUDE in your program's MAP structure.

To INCLUDE the file in the Application Generator:

1. Press the **Global** button.
2. Press the **Embeds** button.
3. Highlight *Inside the Global Map*, then press the Insert button.
4. Highlight *Source*, then press the **Select** button.
5. In the **Source Editor**, type the following:

```
INCLUDE('dde.clw')
```
6. Save and Exit the Source Editor.
7. Press the **Close** button, then press the **Ok** button.

Connect to Clarion as DDE Server

```
Server=DDECLIENT('ClarionWin')
```

Server A LONG variable to hold the DDE client channel number.

DDECLIENT This procedure returns a new DDE client channel number for the *application* and *topic*. If the *application* is not currently executing, DDECLIENT returns zero (0).

ClarionWin The identifier for Clarion.

This connects your client application to the Clarion environment to use it as a server. A connection must be made to the server before any other commands can be issued.

The example below demonstrates how you can determine if CW is running and RUN it if it isn't.

Example:

```
!Embedded Source in Window Event Handling- after generated code, Timer
```

```
Server = DDECLIENT('ClarionWin')
```

```
IF Server                    ! If the server is running
  DO ProcedureReturn        ! Return to caller
ELSIF NOT Flag#             ! Check flag to see if RUN has been issued
  RUN('C5EE')              ! Start it
  Flag# =1                  ! Set flag to show run has been issued
END
```

Disconnect from the Clarion DDE Server

DDECLOSE(*Server*)

DDECLOSE The procedure which terminates the DDE server link.

Server A LONG variable to hold the DDE client channel number.

This terminates the DDE channel between your client application and the Clarion environment server. The connection must be made to the server before any other commands can be issued.

Example:

```
Server = DDECLIENT('ClarionWin')  
  
!...some executable code  
  
DDECLOSE(Server)
```

Export a Dictionary to Text (TXD) format

```
DDEEXECUTE(Server,['ExportDct(DctName,TextFile,ScreenOutput)'])
```

DDEEXECUTE	Send a command string to a DDE server.
<i>Server</i>	A LONG containing the DDE client channel number.
ExportDct	The service of Exporting a Dictionary to Text format
<i>DctName</i>	Clarion Dictionary from which the TXD will export.
<i>TextFile</i>	The file to which the dictionary is exported.
<i>ScreenOutput</i>	If one (1) or TRUE, screen output is displayed, if zero (0) or FALSE, screen display is suppressed. If omitted, screen output is suppressed.

This service allows you to export a dictionary to text format (.TXD). A connection must be made to the server before this (or any other) command can be issued. You can export a TXD file from an open Dictionary.

Example:

```
Server      LONG
DDEErrorMsg CSTRING(300)
DDEErrorNum USHORT

CODE

SYSTEM{PROP:DDETimeout} = 12000      ! Time out after two minutes
DDEEXECUTE(Server,['ExportDct(c:\CLARION5\APPS\Myapp1.DCT,MYTXD.TXD,0)'])
DO CheckDDEError

CheckDDEError  ROUTINE

DDEErrorMsg = ''
err# = ERRORCODE()
! Check for DDE error
IF err# > 600      ! ERRORCODE is DDE related
  IF err# = 603      !! DDEExecute Failed
    DDEREAD(Server, DDE:manual, 'GetErrorNum', DDEErrorNum)
    DDEREAD(Server, DDE:manual, 'GetErrorMsg', DDEErrorMsg)
    MESSAGE('Error ' & DDEErrorNum & ' : ' & CLIP(DDEErrorMsg))
    ELSIF err# = 605! Timeout Error
      MESSAGE('DDE timeout')
  ELSE
    MESSAGE(err#)
  END
END
END
```

Import a Dictionary from Text (TXD) format

```
DDEEXECUTE(Server,[ImportDct(TextFile,DctName,ScreenOutput)])
```

DDEEXECUTE	Send a command string to a DDE server.
<i>Server</i>	A LONG containing the DDE client channel number.
ImportDct	The service of Importing a Dictionary from Text format
<i>TextFile</i>	The file from which the dictionary is imported.
<i>DctName</i>	Clarion Dictionary to which the TXD will import.
<i>ScreenOutput</i>	If one (1) or TRUE, screen output is displayed, if zero (0) or FALSE, screen display is suppressed. If omitted, screen output is suppressed.

This service allows you to import a dictionary from text format (.TXD). A connection must be made to the server before this (or any other) command can be issued. You cannot import a TXD file to an open Dictionary.

Example:

```
Server      LONG
DDEErrorMsg CSTRING(300)
DDEErrorNum USHORT

CODE

SYSTEM{PROP:DDETimeout} = 12000      ! Time out after two minutes
DDEEXECUTE(Server,[ImportDct(MYTXD.TXD,c:\CLARION5\APPS\Myapp1.DCT,0)])
DO CheckDDEError

CheckDDEError  ROUTINE

DDEErrorMsg = ''
err# = ERRORCODE()
! Check for DDE error
IF err# > 600      ! ERRORCODE is DDE related
  IF err# = 603    !! DDEExecute Failed
    DDEREAD(Server, DDE:manual, 'GetErrorNum', DDEErrorNum)
    DDEREAD(Server, DDE:manual, 'GetErrorMsg', DDEErrorMsg)
    MESSAGE('Error ' & DDEErrorNum & ' : ' & CLIP(DDEErrorMsg))
    ELSIF err# = 605! Timeout Error
      MESSAGE('DDE timeout')
  ELSE
    MESSAGE(err#)
  END
END
END
```

Export an Application to Text (TXA) format

DDEEXECUTE(*Server*,*[ExportApp(AppName,TextFile,ScreenOutput)]*)

DDEEXECUTE	Send a command string to a DDE server.
<i>Server</i>	A LONG containing the DDE client channel number.
ExportApp	The service of Exporting an Application to Text format
<i>AppName</i>	Clarion Application from which the TXA will export.
<i>TextFile</i>	The file to which the Application is exported.
<i>ScreenOutput</i>	If one (1) or TRUE, screen output is displayed, if zero (0) or FALSE, screen display is suppressed. If omitted, screen output is suppressed.

This service allows you to export an Application to text format (.TXA). A connection must be made to the server before this (or any other) command can be issued. The service will operate on the specified application regardless of any .APP file that is currently open in the Clarion development environment.

Example:

```

Server          LONG
DDEErrorMsg    CSTRING(300)
DDEErrorNum    USHORT

CODE

SYSTEM{PROP:DDETimeout} = 12000      ! Time out after two minutes
DDEEXECUTE(Server, '[ExportApp(c:\CLARION5\APPS\Myapp1.APP,MyTXA.TXA,0)]')
DO CheckDDEError

CheckDDEError  ROUTINE

DDEErrorMsg = ''
err# = ERRORCODE()
! Check for DDE error
IF err# > 600      ! ERRORCODE is DDE related
  IF err# = 603    !! DDEExecute Failed
    DDEREAD(Server, DDE:manual, 'GetErrorNum', DDEErrorNum)
    DDEREAD(Server, DDE:manual, 'GetErrorMsg', DDEErrorMsg)
    MESSAGE('Error ' & DDEErrorNum & ' : ' & CLIP(DDEErrorMsg))
    ELSIF err# = 605! Timeout Error
      MESSAGE('DDE timeout')
  ELSE
    MESSAGE(err#)
  END
END
END

```

Import an Application from Text (TXA) format

```
DDEEXECUTE(Server,[ImportApp(TextFile,AppName,ScreenOutput,NameClash)])'
```

DDEEXECUTE	Send a command string to a DDE server.
<i>Server</i>	A LONG containing the DDE client channel number.
ImportApp	The service of Importing an Application from Text format
<i>TextFile</i>	The file from which the dictionary is imported.
<i>AppName</i>	Clarion Application to which the TXA will import.
<i>ScreenOutput</i>	If one (1) or TRUE, screen output is displayed, if zero (0) or FALSE, screen display is suppressed. If omitted, screen output is suppressed.
<i>NameClash</i>	Specifies how the Application Generator handles procedure name clashes. Specify "Rename" to automatically rename any procedures which clash with existing procedures. Specify "Replace" to automatically replace existing procedures with procedures of the same name being imported. If omitted, the user is prompted to specify how to handle name clashes.

This service allows you to import an Application from text format (.TXA). A connection must be made to the server before this (or any other) command can be issued. The service will operate on the specified application regardless of any .APP file that is currently open in the Clarion development environment.

Example:

```
Server      LONG
DDEErrorMsg CSTRING(300)
DDEErrorNum USHORT
CODE
SYSTEM{PROP:DDETimeOut} = 12000      ! Time out after two minutes
DDEEXECUTE(Server,'[ImportApp(MYTXA.TXA,c:\CLARION5\APPS\Myapp1.APP,Replace)]')
DO CheckDDEError
CheckDDEError ROUTINE
DDEErrorMsg = ''
err# = ERRORCODE()
! Check for DDE error
IF err# > 600      ! ERRORCODE is DDE related
  IF err# = 603      !! DDEExecute Failed
    DDEREAD(Server, DDE:manual, 'GetErrorNum', DDEErrorNum)
    DDEREAD(Server, DDE:manual, 'GetErrorMsg', DDEErrorMsg)
    MESSAGE('Error ' & DDEErrorNum & ' : ' & CLIP(DDEErrorMsg))
  ELSIF err# = 605! Timeout Error
    MESSAGE('DDE timeout')
  ELSE
    MESSAGE(err#)
  END
END
END
```

Load an Application

DDEEXECUTE(*Server*,[**LoadApplication**(*AppName*)]')

DDEEXECUTE Send a command string to a DDE server.

Server A LONG containing the DDE client channel number.

LoadApplication The service of Loading an Application.

AppName The name of the Clarion Application to load.

This service allows you to load an Application into the Application Generator. A connection must be made to the server before this (or any other) command can be issued.

Example:

```

Server      LONG
DDEErrorMsg CSTRING(300)
DDEErrorNum USHORT

CODE

SYSTEM{PROP:DDETimeout} = 12000      ! Time out after two minutes
DDEEXECUTE(Server, '[LoadApplication(c:\CLARION5\APPS\Myapp.app,0)]')
DO CheckDDEError

CheckDDEError  ROUTINE

DDEErrorMsg = ''
err# = ERRORCODE()
! Check for DDE error
IF err# > 600      ! ERRORCODE is DDE related
  IF err# = 603      !! DDEExecute Failed
    DDEREAD(Server, DDE:manual, 'GetErrorNum', DDEErrorNum)
    DDEREAD(Server, DDE:manual, 'GetErrorMsg', DDEErrorMsg)
    MESSAGE('Error ' & DDEErrorNum & ' : ' & CLIP(DDEErrorMsg))
  ELSIF err# = 605! Timeout Error
    MESSAGE('DDE timeout')
  ELSE
    MESSAGE(err#)
  END
END
END

```

Generate an Application

DDEEXECUTE(*Server*,**[GenerateApp**(*AppName*,*ScreenOutput*)**]**)

DDEEXECUTE	Send a command string to a DDE server.
<i>Server</i>	A LONG containing the DDE client channel number.
GenerateApp	The service of Generating an Application.
<i>AppName</i>	Clarion Application to generate.
<i>ScreenOutput</i>	If one (1) or TRUE, screen output is displayed, if zero (0) or FALSE, screen display is suppressed. If omitted, screen output is suppressed.

This service allows you to generate the source code for an Application. A connection must be made to the server before this (or any other) command can be issued. The service will operate on the specified application regardless of any .APP file that is currently open in the Clarion development environment.

Example:

```

Server      LONG
DDEErrorMsg CSTRING(300)
DDEErrorNum USHORT

CODE

SYSTEM{PROP:DDETimeout} = 12000      ! Time out after two minutes
DDEEXECUTE(Server, '[GenerateAPP(c:\CLARION5\APPS\Myapp.app,0)]')
DO CheckDDEError

CheckDDEError  ROUTINE

DDEErrorMsg = ''
err# = ERRORCODE()
! Check for DDE error
IF err# > 600      ! ERRORCODE is DDE related
  IF err# = 603    !! DDEExecute Failed
    DDEREAD(Server, DDE:manual, 'GetErrorNum', DDEErrorNum)
    DDEREAD(Server, DDE:manual, 'GetErrorMsg', DDEErrorMsg)
    MESSAGE('Error ' & DDEErrorNum & ' : ' & CLIP(DDEErrorMsg))
  ELSIF err# = 605! Timeout Error
    MESSAGE('DDE timeout')
  ELSE
    MESSAGE(err#)
  END
END
END

```

Execute a Project or Application

DDEEXECUTE(*Server*,**[ExecuteProject**(*ProjectName*)**]**)

DDEEXECUTE Send a command string to a DDE server.

Server A LONG containing the DDE client channel number.

ExecuteProject The service of compiling and linking a Project (.PRJ) or Application (.APP).

ProjectName The Clarion Project or Application to execute.

This service allows you to compile and link a Project or Application. A connection must be made to the server before this (or any other) command can be issued. The service will operate on the specified application regardless of any .APP file that is currently open in the Clarion development environment. The screen output is displayed, but the window will close upon completion without any interaction.

Example:

```
Server      LONG
DDEErrorMsg CSTRING(300)
DDEErrorNum USHORT

CODE

SYSTEM{PROP:DDETimeout} = 12000      ! Time out after two minutes
DDEEXECUTE(Server, '[ExecuteProject(c:\CLARION5\APPS\Myapp.app)]')
DO CheckDDEError

CheckDDEError  ROUTINE

DDEErrorMsg = ''
err# = ERRORCODE()
! Check for DDE error
IF err# > 600      ! ERRORCODE is DDE related
  IF err# = 603      !! DDEExecute Failed
    DDEREAD(Server, DDE:manual, 'GetErrorNum', DDEErrorNum)
    DDEREAD(Server, DDE:manual, 'GetErrorMsg', DDEErrorMsg)
    MESSAGE('Error ' & DDEErrorNum & ' : ' & CLIP(DDEErrorMsg))
  ELSIF err# = 605! Timeout Error
    MESSAGE('DDE timeout')
  ELSE
    MESSAGE(err#)
  END
END
END
```

Running a Utility Template

```
DDEEXECUTE(Server,[GenerateUtilityTemplate(UtilTemplateName,AppName,ScreenOutput)])
```

DDEEXECUTE Send a command string to a DDE server.

Server A LONG containing the DDE client channel number.

GenerateUtilityTemplate

The service of Generating a Utility Template on the specified Application.

UtilTemplateName The Utility template to generate.

AppName The Clarion Application to which the Utility Template is generated.

ScreenOutput If one (1) or TRUE, screen output is displayed, if zero (0) or FALSE, screen display is suppressed. If omitted, screen output is suppressed.

This service allows you to run a Utility template on an Application. A connection must be made to the server before this (or any other) command can be issued. The service will operate on the specified application regardless of any .APP file that is currently open in the Clarion development environment.

Example:

```
Server      LONG
DDEErrorMsg CSTRING(300)
DDEErrorNum USHORT

CODE
SYSTEM{PROP:DDETimeout} = 12000      ! Time out after two minutes
DDEEXECUTE(Server,[GenerateUtilityTemplate(MyUtil,c:\CLARION5\APPS\Myapp.APP,0)])
DO CheckDDEError

CheckDDEError  ROUTINE

DDEErrorMsg = ''
err# = ERRORCODE()
! Check for DDE error
IF err# > 600      ! ERRORCODE is DDE related
  IF err# = 603    !! DDEExecute Failed
    DDEREAD(Server, DDE:manual, 'GetErrorNum', DDEErrorNum)
    DDEREAD(Server, DDE:manual, 'GetErrorMsg', DDEErrorMsg)
    MESSAGE('Error ' & DDEErrorNum & ' : ' & CLIP(DDEErrorMsg))
    ELSIF err# = 605! Timeout Error
      MESSAGE('DDE timeout')
  ELSE
    MESSAGE(err#)
  END
END
END
```

Registering a Template Class

DDEEXECUTE(*Server*,**[RegisterTemplateChain**(*TemplateName*,*ScreenOutput*)]')

DDEEXECUTE Send a command string to a DDE server.

Server A LONG containing the DDE client channel number.

RegisterTemplateChain

The service of registering a specified template set.

TemplateName The name of the template set to register.

ScreenOutput If one (1) or TRUE, screen output is displayed, if zero (0) or FALSE, screen display is suppressed. If omitted, screen output is suppressed.

This feature is primarily intended for add-on products for Clarion. By allowing a template set to be registered by a client application, template developers can add this functionality to the installation procedure. This eliminates the need for end users to register add-on template sets.

Example:

```
Server      LONG
DDEErrorMsg CSTRING(300)
DDEErrorNum USHORT

CODE

SYSTEM{PROP:DDETimeout} = 12000      ! Time out after two minutes
DDEEXECUTE(Server, '[RegisterTemplateChain(MYTPL.TPL,0)]')
D0 CheckDDEError

CheckDDEError  ROUTINE

DDEErrorMsg = ''
err# = ERRORCODE()
! Check for DDE error
IF err# > 600      ! ERRORCODE is DDE related
  IF err# = 603      !! DDEExecute Failed
    DDEREAD(Server, DDE:manual, 'GetErrorNum', DDEErrorNum)
    DDEREAD(Server, DDE:manual, 'GetErrorMsg', DDEErrorMsg)
    MESSAGE('Error ' & DDEErrorNum & ' : ' & CLIP(DDEErrorMsg))
  ELSIF err# = 605! Timeout Error
    MESSAGE('DDE timeout')
  ELSE
    MESSAGE(err#)
  END
END
```

Getting DDE Error Messages

```
DDEREAD(Server,DDE:Manual,'GetErrorMsg',ErrorMessageText)
DDEREAD(Server,DDE:Manual,'GetErrorNum',DDEErrorNum)
```

DDEREAD	Get data from the DDE server.
<i>Server</i>	A LONG containing the DDE client channel number.
DDE:Manual	An EQUATE defining the type of data link as manual (defined in EQUATES.CLW).
'GetErrorMsg'	Request for the error message data item from the server. This must be provided exactly as shown.
'GetErrorNum'	Request for the error number data item from the server. This must be provided exactly as shown.
<i>ErrorMessageText</i>	A STRING(255) variable to which the error message text is assigned.
<i>DDEErrorNum</i>	A USHORT variable to which the error number is assigned.

This feature is provided to allow error checking in your DDE client application. You can check the Clarrion ERRORCODE to determine if the error posted is DDE-related, then query the DDE server and evaluate the DDE error returned.

Example:

```
Server      LONG
DDEErrorMsg CSTRING(300)
DDEErrorNum USHORT

CODE
DDEEXECUTE(Server, '[GenerateAPP(c:\CLARION5\APPS\Myapp.app,0)]')
DO CheckDDEError

CheckDDEError  ROUTINE

DDEErrorMsg = ''
err# = ERRORCODE()
! Check for DDE error
IF err# > 600      ! ERRORCODE is DDE related
  IF err# = 603    !! DDEExecute Failed
    DDEREAD(Server, DDE:manual, 'GetErrorNum', DDEErrorNum)
    DDEREAD(Server, DDE:manual, 'GetErrorMsg', DDEErrorMsg)
    MESSAGE('Error ' & DDEErrorNum & ' : ' & CLIP(DDEErrorMsg))
    ELSIF err# = 605! Timeout Error
      MESSAGE('DDE timeout')
  ELSE
    MESSAGE(err#)
  END
END
```

Clarion DDE Errors

601 Invalid DDE Channel
602 DDE Channel Not Open
603 DDEEXECUTE Failed
604 DDEPOKE Failed
605 Time Out

DDE Service Errors and associated messages

1 "Screen output must be either 'TRUE' or 'FALSE'"
2 "Cannot open specified dictionary"
3 "Cannot create specified text file"
4 "Cannot open specified text file"
5 "Cannot create specified dictionary"
6 "Cannot open specified application"
7 "Cannot create specified application"
8 "Cannot open specified template file"
9 "Cannot open specified project file"
10 "Wrong number of parameters"
11 "Requested DDE service is not supported"

The following errors return a dynamic message generated by the requested service:

12 Export dictionary failed
13 Import dictionary failed
14 Export application failed
15 Import application failed
16 Register templated chained failed
17 Generate utility template failed
18 Generate application failed
19 Error in Make

10 - .TXD FILE FORMAT

.TXD Files: Clarion Dictionaries in ASCII Format

.TXD files are simply ASCII text versions of Clarion .DCT files. .DCT files are where the Clarion environment stores all the reusable information about a database. A .TXD file contains all the information stored in an .DCT file (except for prior versions), but in a text format that is readable (and writable) by most text editors.

You can easily create a .TXD file for any .DCT file by loading the dictionary into the Clarion development environment, then choosing **File ► Export Text** from the menu.

Why would you need a .TXD file? For at least three reasons:

- | | |
|--------------|---|
| Backup | Because a .TXD file can be imported just as easily as it is exported, it may serve as a backup of the current version of your .DCT file. Previous versions are not available in the .TXD. |
| Mass Changes | Because a .TXD file may be manipulated in your favorite text editor, you can use the power of the text editor to make mass changes to your dictionary, or for that matter, any changes that would be easier with a text editor. |
| First Aid | Occasionally, a .DCT file may exhibit some strange behavior in the development environment. Exporting to a .TXD file, then importing from that same file can, highlight problems which will enable you to correct any problems in the dictionary. |

.TXD File Organization

There are four major sections in the .TXD file: [DICTIONARY], [FILES], [ALIASES], and [RELATIONS]. These sections correspond to the main Clarion language database structures, and they contain subsections and keywords that fully describe these structures, for example, the [FILES] section contains, among other things, the field and key definitions for the file.

.TXD Skeleton

The following is an ordered list of .TXD sections, subsections, and keywords. This list is designed to give you a feel for the overall structure and organization of the .TXD file.

Each section begins with its title enclosed in square brackets and ends with the beginning of another section. Each section may contain subsections and keywords.

Subsections begin just like the major sections—with its title surrounded by square brackets.

Keywords appear as a keyword name in all capitals followed by the keyword value. The keyword values appear in various formats described below on a case by case basis.

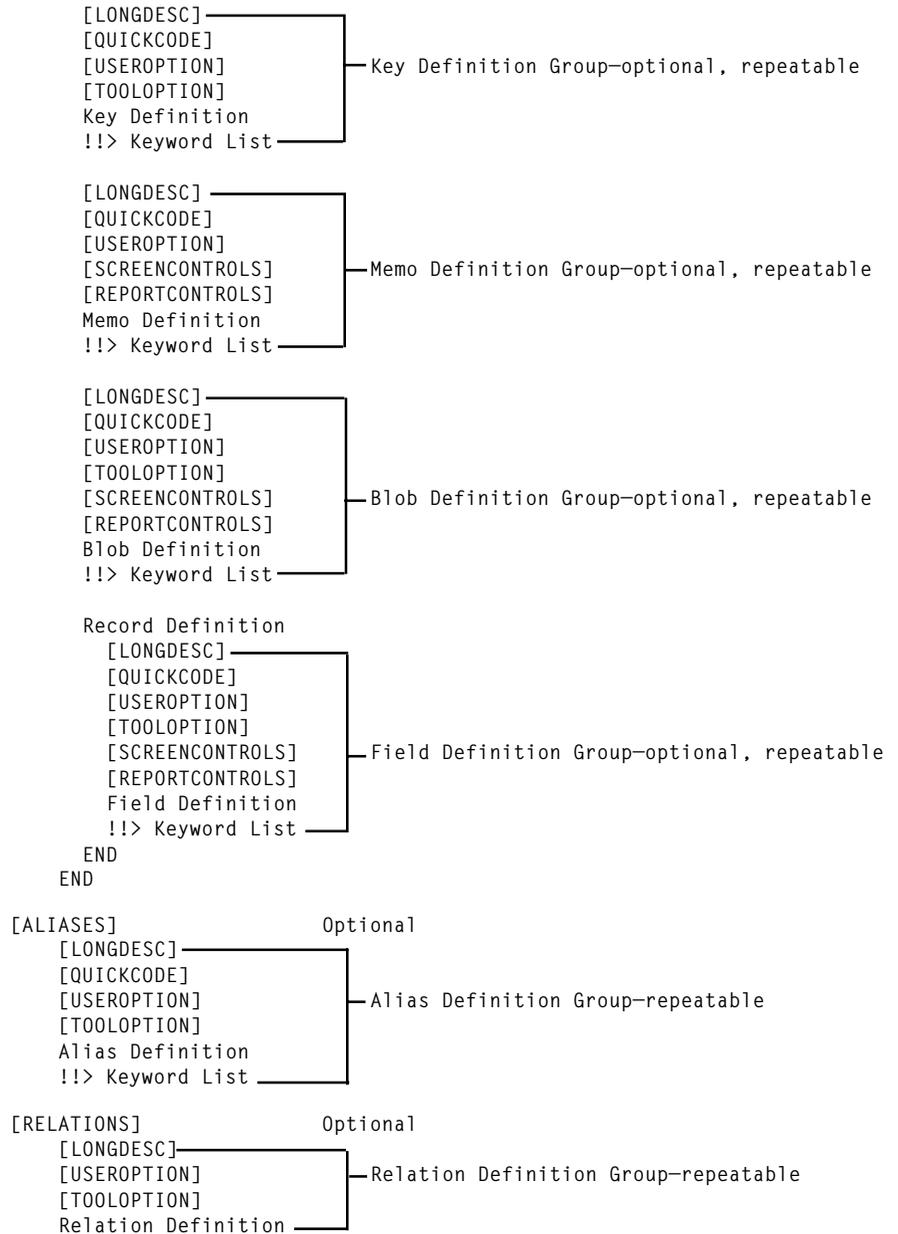
The spacing and indentations in the list are for readability and do not appear in the actual .TXD files.

Following the skeleton is a detailed discussion of each .TXD section, subsection, and keyword.

```
[DICTIONARY]
  VERSION
  CREATED
  MODIFIED
  PASSWORD
  [DESCRIPTION]
  [TOOLOPTION]

[FILES]
  [LONGDESC]
  [QUICKCODE]
  [USEROPTION]
  [TOOLOPTION]
  File Definition
  !!> Keyword List
```

File Definition Group—repeatable



.TXD File Sections

[DICTIONARY]

The dictionary section is required, although all of its individual keywords and subsections are optional. The dictionary section contains the following keywords and subsections:

```
[DICTIONARY]
  VERSION
  CREATED
  MODIFIED
  PASSWORD
  [DESCRIPTION]
[TOOLOPTION]
```

VERSION	Lists the version number of the dictionary (optional). Version is ignored on import of .TXD. For example: VERSION '1.0'
CREATED	Lists the date and time the dictionary was originally created (optional). For example: CREATED '19 AUG 95' '11:02:33AM'
MODIFIED	Lists the date and time the dictionary was last saved (optional). For example: MODIFIED ' 7 NOV 95' ' 9:44:18AM'
PASSWORD	Lists the password needed to access the dictionary (optional). Use the Password button in the Dictionary Properties dialog to establish the password for your dictionary.

The password is not case sensitive. The password is encrypted in the .DCT file, but is not encrypted when exported to the .TXD file. For example:

```
PASSWORD cablecar
```

[DESCRIPTION]	Lists up to 1000 characters of descriptive text on multiple lines (optional). Each line of text begins with an exclamation point (!) and contains up to 75 characters. The text comes from the Comments tab in the Dictionary Properties dialog.
----------------------	--

A [DESCRIPTION] will be split into lines of 75 characters each in the .TXD. If the text is separated by a carriage return (<CR>) then it will write out an extra empty line. This is true for any description ([DESCRIPTION] and [LONGDESC]) within the .TXD and for variables in the .TXA. For example:

Original text in the dictionary:

*This is a description : <CR>
and it continues on the next line whereby the text
from now on is written without carriage returns.*

Text in the .TXD:

```
!This is a description :
!  
!and it continues on the next line whereby the text from now on is
!written without carriage returns.
```

[TOOLOPTION] See *Common Subsections* below (optional).

Example—[DICTIONARY]

```
[DICTIONARY]
VERSION '1.0'
CREATED '19 AUG 95' '11:02:33AM'
MODIFIED ' 7 NOV 95' ' 9:44:18AM'
PASSWORD cablecar
[DESCRIPTION]
!Up to 1000 characters of text describing this dictionary.
!Each line of up to 75 characters begins with an exclamation point.
```

[FILES]

The files section appears only once in the .TXD file. All files in the dictionary are defined in this section. Fields and keys are also defined in this section as an integral part of the definition of each file.

File Definition Group

The file definition group is a series of .TXD subsections and keywords that fully describes a single FILE within the data dictionary. The group is repeated for every FILE in the dictionary, so that all are fully documented. The file definition group contains the following keywords and subsections:

```
[FILES]
[LONGDESC]
[QUICKCODE]
[USEROPTION]
[TOOLOPTION]
File Definition Group-repeatable
File Definition
!!> Keyword List
```

[LONGDESC] See *Common Subsections* below (optional).

[QUICKCODE] Information used by the Clarion Wizards to configure your Wizard generated applications and procedures (optional). Use the **Options** tab of the **File Properties** dialog to input the [QUICKCODE] information.

The only Wizard/QUICKCODE option available for files is NOPOPULATE. NOPOPULATE means the Wizards will not generate a form, a browse, or a report procedure for this file.

[USEROPTION] See *Common Subsections* below (optional).

[TOOLOPTION] See *Common Subsections* below (optional).

File Definition Begins with the first line of the Clarion language FILE declaration statements for this file (required). Optionally includes a comment up to 40 characters long. The file definition also includes the keyword list, plus the key, memo, blob, record, and field definition groups. The entire structure ends with END. For example:

```

Customers FILE,DRIVER('TOPSPEED'),PRE(CUS),CREATE
  Key Definition Group
  Memo Definition Group
  Blob Definition Group
  Record Definition Group
  Field Definition Group
END

```

Keyword List	A list of internal keywords that describe options not specified on the FILE statement (optional). The list begins with !!>.
IDENT()	The internal reference number the development environment uses to identify the FILE (optional). !!> IDENT(1)
USAGE()	Contains either FILE, GLOBAL, or POOL to identify whether the FILE is global data or a field pool (optional). !!> IDENT(1),USAGE(POOL)
LAST	Specifies USAGE(GLOBAL) data which should generate last (optional). !!> IDENT(1),USAGE(GLOBAL),LAST

Example—File Definition Group

```

[FILES]

[LONGDESC]
!This is the main customer file. Contains names addresses and
!phone numbers. One record per customer. Each customer has a unique
!key that is auto numbered...
[QUICKCODE]
!NOPOPULATE
[USEROPTION]
!ThirdPartyTemplateSwitch(on)
!ThirdPartyPreProcessLevel(release)
Customers FILE,DRIVER('TOPSPEED'),PRE(CUS),CREATE,THREAD
!!> IDENT(1)
.
.
.
END

```

Key Definition Group

The key definition group is optional. It is a series of .TXD subsections and keywords that fully describes a single KEY or INDEX for a file. The group is repeated for every KEY or INDEX to the file, so that all are fully documented. The key definition group contains the following keywords and subsections:

```
[LONGDESC]
[QUICKCODE]
[USEROPTION]
[TOOLOPTION]
Key Definition Group-optional, repeatable
Key Definition
!!> Keyword List
```

[LONGDESC] See *Common Subsections* below (optional).

[QUICKCODE] Information used by the Clarion Wizards to configure your Wizard generated applications and procedures (optional). Use the **Options** tab of the **Key Properties** dialog to input the [QUICKCODE] information.

[QUICKCODE] information may be specified on multiple lines, each line beginning with an exclamation point. However, multiple lines are concatenated to one string by the development environment. Therefore keywords should be separated by a comma, even when on different lines in the .TXD.

The Wizard/QUICKCODE options available for keys are NOPOPULATE and ORDER:

NOPOPULATE The Wizards do not generate reports or browses sorted on this key (optional).

ORDER() The order in which browses and reports sorted on this key appear within Wizard generated menus and browse boxes (optional).

The order may be specified as **Normal**, **First**, or **Last**. Normal displays items in the order in which they are found in the dictionary. First forces the item to appear before all Normal and Last items. Conversely, Last forces the item to appear after all First and Normal items. For example:

```
[QUICKCODE]
!ORDER(First)
```

[USEROPTION] See *Common Subsections* below (optional).

[TOOLOPTION] See *Common Subsections* below (optional).

KEY Definition Lists the Clarion language KEY or INDEX declaration (required). Optionally includes a comment up to 40 characters long. For example:

	KeyCust KEY(CUS:CustNumber),NOCASE !Auto-number
Keyword List	A list of internal keywords that describe options not specified on the KEY statement (optional). The list begins with !!>. The two valid keywords for keys are IDENT and AUTO.
IDENT()	The internal reference number the development environment uses to identify the KEY (optional).
AUTO	The key is auto-numbered (optional). This means the application generator will supply code to automatically increment the value of the key. Use the Attributes tab of the Key Properties dialog to set the auto-number keyword. For example: !!> IDENT(1),AUTO

Example—Key Definition Group

```
KeyCustNumber KEY(CUS:CustNumber),NOCASE,OPT !Auto-numbered Cust Key
!!> IDENT(1),AUTO
[QUICKCODE]
!ORDER(First)
```

Memo Definition Group

The memo definition group is optional. It is a series of .TXD subsections and keywords that fully describes a single MEMO field in a file. The group is repeated for each MEMO field in the file, so that all are fully documented. The memo definition group contains the following keywords and subsections:

```
[LONGDESC]
[QUICKCODE]
[USEROPTION]
[TOOLOPTION]
[SCREENCONTROLS] Memo Definition Group—optional, repeatable
[REPORTCONTROLS]
MEMO Definition
!!> Keyword List
```

[LONGDESC]	See <i>Common Subsections</i> below (optional).
[QUICKCODE]	Information used by the Clarion Wizards to configure your Wizard generated applications and procedures (optional). Use the Options tab of the Field Properties dialog to input the [QUICKCODE] information.

[QUICKCODE] information may be specified on multiple lines, each line beginning with an exclamation point. However, multiple lines are concatenated to one string by the development environment. Therefore keywords should be separated by a comma, even when on different lines in the .TXD.

Wizard/QUICKCODE options available for MEMOs are NOPOPULATE, ORDER(), VERTICALSPACE, and TAB():

NOPOPULATE This field is not included on Wizard generated reports, forms, and browses (optional).

ORDER() Not meaningful for MEMOs. MEMOs always appear alone on a separate tab, in the order in which they appear in the dictionary.

VERTICALSPACE Not meaningful for MEMOs. MEMOs always appear alone on a separate tab.

TAB() Not meaningful for MEMOs. MEMOs always appear alone on a separate tab.

[QUICKCODE]
!NOPOPULATE

[USEROPTION] See *Common Subsections* below (optional).

[TOOLOPTION] See *Common Subsections* below (optional).

[SCREENCONTROLS] MEMOs always appear in TEXT controls. See *Common Subsections* below (optional).

[REPORTCONTROLS] MEMOs always appear in TEXT controls. See *Common Subsections* below (optional).

MEMO Definition The Clarion language MEMO declaration (required). Optionally includes a comment up to 40 characters long. For example:

```
CustomerMemo MEMO(500) !500 character text field
```

Keyword List—MEMOs

A list of internal keywords that describe options not specified by the Clarion language MEMO statement. The list begins with !!> and is optional. Within the .TXD, the keyword list appears on a single line, with keywords separated by commas.

The keywords in this list are set using the **Field Properties** dialog. Many of the keywords correspond directly to Clarion language keywords. See the *Language Reference* for more information on these keywords.

Following is a complete list of keywords available for memos:

```
IDENT()
VALID()
INITIAL()
PROMPT()
HEADER()
HELP()
MESSAGE()
TOOLTIP()
PICTURE()
CASE()
TYPEMODE()
PASSWORD
READONLY
```

Notice that the keywords in the list correspond closely to the tabs and prompts in the **Field Properties** dialog. This is because the **Field Properties** dialog is where these values are set (see the *User's Guide*, the *Language Reference*, and the on-line help for more information on these fields).

IDENT() The internal ID number that the development environment uses to reference this item (optional). For example:

```
IDENT(3)
```

VALID() Specifies the validity checking code that is generated for this field (optional). The **VALID** keyword has a significant parameter list which is diagrammed as follows, where the vertical list represents alternative parameters and the curly braces represent optional parameters:

```
VALID( NONZERO      )
      INRANGE({minimum}{,maximum})
      BOOLEAN
      INFILE(filename,parentfile:childfile)
      INLIST('item1|item2|...|itemn')

```

For example, a required field has:

```
VALID(NONZERO)
```

A non-negative numeric field has:

```
VALID(INRANGE(0))
```

A binary logical field (yes/no, true/false) has:

```
VALID(BOOLEAN)
```

A field validated against another file has:

```
VALID(INFILE(States,Customers:States))
```

A field validated against a fixed, finite list has:

```
VALID(INLIST('Left|Center|Right'))
```

The **Validity Checks** tab in the **Field Properties** dialog specifies these values. See *Using the Dictionary Editor* in the *User's Guide* for more information on validity checks.

INITIAL()

The initial value of the field (optional). The **Attributes** tab in the **Field Properties** dialog sets this value. For example:

```
INITIAL(0)
```

PROMPT()

The text string used as the default prompt for this field on a screen (optional). The **General** tab in the **Field Properties** dialog sets this value. For example:

```
PROMPT('&Product Number')
```

HEADER()

The text string used as the default column title for reports and list boxes (optional). This value is set on the **General** tab of the **Field Properties** dialog. For example:

```
HEADER('PRODUCT NUMBER')
```

HELP()

The help topic for this field. This value is set on the **Help** tab of the **Field Properties** dialog. For example:

```
HELP('PRODNUMBER')
```

MESSAGE()

Up to 75 characters of default message text for this field (optional). This value is set on the **Help** tab of the **Field Properties** dialog. For example:

```
MESSAGE('Enter the 6 digit product number')
```

TOOLTIP()

Up to 75 characters of default tool tip (balloon help) text for this field (optional). This value is set on the **Help** tab of the **Field Properties** dialog. For example:

```
TOOLTIP('Enter the 6 digit product number')
```

PICTURE()	The default screen picture token for this field (optional). See the <i>Language Reference</i> for a complete list of picture tokens and their uses. This value is set on the General tab of the Field Properties dialog. For example: PICTURE('@n6')
CASE()	The default case translation for this field (optional). The choices are UPPER and CAPS. UPPER converts all text to uppercase. CAPS converts all text to mixed case word capitalization. This value is set on the Attributes tab of the Field Properties dialog. For example: CASE('UPPER')
TYPEMODE()	The default typing mode for this field (optional). The choices are INS and OVR. INS preserves existing text by inserting new characters. OVR discards existing text by overwriting it with new text. This value is set on the Attributes tab of the Field Properties dialog. For example: TYPEMODE('INS')
PASSWORD	Text typed in this field is displayed as asterisks (optional). This value is set on the Attributes tab of the Field Properties dialog. For example: PASSWORD
READONLY	The field is read only and therefore will not accept input (optional). This value is set on the Attributes tab of the Field Properties dialog. For example: READONLY

For example:

```
!!> IDENT(47),PROMPT('Customer Memo:'),PICTURE(@s255)
```

Example—Memo Definition Group

```
[LONGDESC]
!Long description of Memo field
[QUICKCODE]
!TAB('TabName'),ORDER(Last),VERTICALSPACE
[SCREENCONTROLS]
! PROMPT('Customer Memo:'),USE(?CUS:CustomerMemo:Prompt)
! TEXT,USE(CUS:CustomerMemo)
[REPORTCONTROLS]
! TEXT,USE(CUS:CustomerMemo)
CustomerMemo          MEMO(500)
!!> IDENT(47),PROMPT('Customer Memo:'),PICTURE(@s255)
```

Blob Definition Group

The blob definition group is optional. It is a series of .TXD subsections and keywords that fully describes a single BLOB field in a file. The group is repeated for each BLOB field in the file, so that all are fully documented. The blob definition group contains the following keywords and subsections:

```
[LONGDESC]
[QUICKCODE]
[USEROPTION]
  [TOOLOPTION]
[SCREENCONTROLS]          Blob Definition Group-optional, repeatable
[REPORTCONTROLS]
BLOB Definition
!!> Keyword List
```

[LONGDESC] See *Common Subsections* below (optional).

[QUICKCODE] Not meaningful for BLOBs.

[USEROPTION] See *Common Subsections* below (optional).

[TOOLOPTION] See *Common Subsections* below (optional).

[SCREENCONTROLS]
See *Common Subsections* below (optional).

[REPORTCONTROLS]
See *Common Subsections* below (optional).

BLOB Definition Lists the Clarion language BLOB declaration (required). Optionally includes a comment up to 40 characters long. For example:

```
CustomerPhoto BLOB,BINARY !Customer Image/Logo
```

Keyword List A list of internal keywords that describe options not specified on the BLOB statement (optional). The list begins with !!>.

The only valid keyword for a BLOBs is IDENT. IDENT supplies the internal reference number by which the development environment identifies the FILE. For example:

```
!!> IDENT(48)
```

Example—Blob Definition Group

```
[LONGDESC]
!This digitized image may contain customer logo or photograph
[QUICKCODE]
```

```

!TAB('Personal'),ORDER(First)
[SCREENCONTROLS]
! IMAGE,USE(?CUS:CustomerPhoto)
[REPORTCONTROLS]
! IMAGE,USE(?CUS:CustomerPhoto)
CustomerPhoto          BLOB,BINARY !Customer Image/Logo
!!> IDENT(48)

```

Record Definition

The record definition is required. It begins with the Clarion language RECORD declaration statements for this file. The RECORD definition then includes all the field definition groups for the file. The entire structure ends with END. For example:

```

CustomerRecord    RECORD

    field definition groups

END

```

Field Definition Group

The field definition group is optional. It is a series of .TXD subsections and keywords that fully describes a single field in a file. The group is repeated for each field in the file, so that all are fully documented. The field definition group contains the following keywords and subsections:

```

[LONGDESC]
[QUICKCODE]
[USEROPTION]
    [TOOLOPTION]
[SCREENCONTROLS]          Field Definition Group—optional, repeatable
[REPORTCONTROLS]
Field Definition
!!> Keyword List

```

[LONGDESC] See *Common Subsections* below (optional).

[QUICKCODE] Information used by the Clarion Wizards to configure your Wizard generated applications and procedures (optional). Use the **Options** tab of the **Field Properties** dialog to input the [QUICKCODE] information.

[QUICKCODE] information may be specified on multiple lines, each line beginning with an exclamation point. However, multiple lines are concatenated to one string by the development environment. Therefore keywords should be separated by a comma, even when on different lines in the .TXD.

The Wizard/QUICKCODE options available for fields are NOPOPULATE, ORDER, VERTICALSPACE, and TAB:

NOPOPULATE The Wizards do not include this field on browses, forms, and reports (optional).

ORDER() The order in which this field is populated on Wizard generated browses, forms, and reports (optional).

The order may be specified as **Normal**, **First**, or **Last**. Normal displays items in the order in which they are found in the dictionary. First forces the item to appear before all Normal and Last items. Conversely, Last forces the item to appear after all First and Normal items.

VERTICALSPACE

Extra space is inserted between this field and the preceding field on Wizard generated forms.

TAB() The name of the TAB on which this field appears on Wizard generated forms.

For example:

```
[QUICKCODE]
!ORDER(Last),VERTICALSPACE,TAB('Personal')
```

[USEROPTION] See *Common Subsections* below (optional).

[TOOLOPTION] See *Common Subsections* below (optional).

[SCREENCONTROLS] See *Common Subsections* below (optional).

[REPORTCONTROLS] See *Common Subsections* below (optional).

Field Definition The Clarion language field declaration statement (required). Optionally includes a comment up to 40 long. For example:

```
CustNumber DECIMAL(7,2) !unique key
```

Keyword List—Fields

A list of internal keywords that describe options not specified by the Clarion language field declaration statement. The list begins with !> and is optional. Within the .TXD, the keyword list appears on a single line, with keywords separated by commas.

The keywords in this list are set using the **Field Properties** dialog. Many of the keywords correspond directly to Clarion language keywords. See the *Language Reference* for more information on these keywords.

Following is a complete list of keywords available for data dictionary fields.:

```
IDENT()
VALID()
VALUES()
INITIAL()
PROMPT()
HEADER()
HELP()
MESSAGE()
TOOLTIP()
PICTURE()
CASE()
TYPEMODE()
PASSWORD
READONLY
```

Notice that the keywords in the list correspond closely to the tabs and prompts in the **Field Properties** dialog. This is because the **Field Properties** dialog is where these values are set (see the *User's Guide*, the *Language Reference*, and the on-line help for more information these fields).

IDENT() The internal ID number that the development environment uses to reference this item (optional). For example:

```
IDENT(3)
```

VALID() Specifies the validity checking code that is generated for this field (optional). The **VALID** keyword has a significant parameter list which is diagrammed as follows, where the vertical list represents alternative parameters and the curly braces represent optional parameters:

```
VALID( NONZERO      )
      INRANGE({minimum}{,maximum])
      BOOLEAN
      INFILE(filename,parentfile:childfile)
      INLIST('item1|item2|...|itemn')
      NOCHECKS('item1|item2|...|itemn')
```

For example, a required field has:

```
VALID(NONZERO)
```

A non-negative numeric field has:

```
VALID(INRANGE(0))
```

A binary logical field (yes/no, true/false) has:

VALID(BOOLEAN)

A field validated against another file has:

VALID(INFILE(States,Customers:States))

A field validated against a fixed, finite list has:

VALID(INLIST('Left|Center|Right'))

The **Validity Checks** tab in the **Field Properties** dialog specifies these values. See *Using the Dictionary Editor* in the *User's Guide* for more information on validity checks.

VALUES()

Specifies the displayed list of selections for validity checking code that is generated for this field (optional). This is related to the VALID keyword and is active for NOCHECKS, INLIST, and BOOLEAN. The **Validity Checks** tab in the **Field Properties** dialog specifies these values. See *Using the Dictionary Editor* in the *User's Guide* for more information on validity checks.

INITIAL()

The initial value of the field (optional). The **Attributes** tab in the **Field Properties** dialog sets this value. For example:

INITIAL(0)

PROMPT()

The text string used as the default prompt for this field on a screen (optional). The **General** tab in the **Field Properties** dialog sets this value. For example:

PROMPT('&Product Number')

HEADER()

The text string used as the default column title for reports and list boxes (optional). This value is set on the **General** tab of the **Field Properties** dialog. For example:

HEADER('PRODUCT NUMBER')

HELP()

The help topic for this field. This value is set on the **Help** tab of the **Field Properties** dialog. For example:

HELP('PRODNUMBER')

MESSAGE()

Up to 75 characters of default message text for this field (optional). This value is set on the **Help** tab of the **Field Properties** dialog. For example:

MESSAGE('Enter the 6 digit product number')

TOOLTIP()	Up to 75 characters of default tool tip (balloon help) text for this field (optional). This value is set on the Help tab of the Field Properties dialog. For example: TOOLTIP('Enter the 6 digit product number')
PICTURE()	The default screen picture token for this field (optional). See the <i>Language Reference</i> for a complete list of picture tokens and their uses. This value is set on the General tab of the Field Properties dialog. For example: PICTURE('@n6')
CASE()	The default case translation for this field (optional). The choices are UPPER and CAPS. UPPER converts all text to uppercase. CAPS converts all text to mixed case word capitalization. This value is set on the Attributes tab of the Field Properties dialog. For example: CASE('UPPER')
TYPEMODE()	The default typing mode for this field (optional). The choices are INS and OVR. INS preserves existing text by inserting new characters. OVR discards existing text by overwriting it with new text. This value is set on the Attributes tab of the Field Properties dialog. For example: TYPEMODE('INS')
PASSWORD	Text typed in this field is displayed as asterisks (optional). This value is set on the Attributes tab of the Field Properties dialog. For example: PASSWORD
READONLY	The field is read only and therefore will not accept input (optional). This value is set on the Attributes tab of the Field Properties dialog. For example: READONLY

For example:

```
!!> IDENT(5),PROMPT('&Cust Number:'),PICTURE(@n4)
```

Example—Field Definition Group

```
[QUICKCODE]
!ORDER(First)
[SCREENCONTROLS]
! PROMPT('&Cust Number:'),USE(?CUS:CustNumber:Prompt)
! ENTRY(@n4),USE(CUS:CustNumber)
```

```
[REPORTCONTROLS]
! STRING(@n4),USE(CUS:CustNumber)
CustNumber          DECIMAL(7,2)
!!> IDENT(5),PROMPT('&Cust Number:'),HEADER('Cust Number'),PICTURE(@n4)
```

[ALIASES]

The aliases section appears once in the .TXD file. This section is optional, and all aliases in the dictionary are defined here.

Alias Definition Group

The alias definition group is a series of .TXD subsections and keywords that fully describes a single alias within the data dictionary. The group is repeated for every alias in the dictionary. The alias definition group contains the following keywords and subsections:

```
[ALIASES]
[LONGDESC]
[QUICKCODE]
[USEROPTION]
[TOOLOPTION]
Alias Definition Group—repeatable
Alias Definition
!!> Keyword List
```

[LONGDESC] See *Common Subsections* below (optional).

[QUICKCODE] Information used by the Clarion Wizards to configure your Wizard generated applications and procedures (optional). Use the **Options** tab of the **Alias Properties** dialog to input the [QUICKCODE] information.

The only Wizard/QUICKCODE option available for aliases is NOPOPULATE. NOPOPULATE means the Wizards will not generate a form, a browse, or a report procedure for this alias.

[USEROPTION] See *Common Subsections* below (optional).

[TOOLOPTION] See *Common Subsections* below (optional).

Alias Definition Specifies the file the alias is redefining, and the new prefix for the alias (required). Optionally includes a comment up to 40 characters long. For example:

```
Sales    ALIAS(Orders),PRE(SAL) !Alias for Order
```

Keyword List A list of internal keywords that describe options not specified in the alias definition (optional). The list begins with !!>.

The only valid keyword for an alias is IDENT. IDENT supplies the internal reference number by which the development environment identifies the alias. For example:

```
!!> IDENT(6)
```

Example—Alias Definition Group

```
[ALIASES]

[QUICKCODE]
!NOPOPULATE
Sales      ALIAS(Orders),PRE(SAL) !Alias for Order
!!> IDENT(6)
```

[RELATIONS]

The relations section appears once in the .TXD file. This section is optional, and all file relationships in the dictionary are defined here. Referential Integrity constraints are also defined here as an integral part of each relationship.

Relation Definition Group

The relation definition group is a series of .TXD subsections, keywords, and a special relation definition that fully describes a relationship between two files in the data dictionary. The group is repeated for each relationship in the dictionary. The relation definition group contains the following keywords and subsections:

```
[RELATIONS]
[LONGDESC]
[USEROPTION]
[TOOLOPTION]
Relation Definition Group—repeatable      Relation Definition
```

[LONGDESC] See *Common Subsections* below (optional).

[USEROPTION] See *Common Subsections* below (optional).

[TOOLOPTION] See *Common Subsections* below (optional).

Relation Definition A relationship defined in the **Edit Relationship Properties** dialog (required). The .TXD relation definition is diagrammed as follows, where the vertical lists represent alternative parameters and the curly braces enclose optional parameters:

```

RELATION, ONE:MANY, UPDATE( RESTRICT ),DELETE(RESTRICT )
      MANY:ONE          CASCADE          CASCADE
                        CLEAR            CLEAR

filename FILE( {key} )
filename RELATED_FILE( {key} )
{FILE_TO_RELATED_KEY
  FIELD( filefieldname,relatedfilefieldname )      repeatable
        NOLINK,          NOLINK
END}
{RELATED_FILE_TO_KEY
  FIELD( relatedfilefieldname,filefieldname )      repeatable
        NOLINK,          NOLINK
END}
END

```

RELATION The beginning of the relationship's definition.

ONE:MANY | MANY:ONE

The type of relationship (required).

UPDATE

The action taken to maintain database referential integrity when the parent file's linking key field values are changed (optional). The action is either RESTRICT, CASCADE, or CLEAR.

RESTRICT

The update is not allowed if there are related child records.

CASCADE

The update is cascaded to the linking key fields of all related child records.

CLEAR

The update is cascaded in the form of blanks, zeros, or nulls to the linking key fields of all related child records.

DELETE

The action taken to maintain database referential integrity when the parent file's record is deleted (optional). The action is either RESTRICT, CASCADE, or CLEAR.

RESTRICT

The delete is not allowed if there are related child records.

CASCADE The delete is allowed, and the related child records are deleted too.

CLEAR The delete is cascaded in the form of blanks, zeros, or nulls to the linking key fields of all related child records.

***filename* FILE({*key*})**

The key in the first file in the relationship that contains the field(s) common to both files (optional). This file is either the parent in a 1:MANY relationship, or the child in a MANY:1 relationship. *key* is optional for a “one-way” or “lookup” relationship.

***filename* RELATED_FILE({*key*})**

The key in the second file in the relationship that contains the field(s) common to both files (optional). This file is either the child in a 1:MANY relationship, or the parent in a MANY:1 relationship. *key* is optional for a “one-way” or “lookup” relationship.

```
RELATION, ONE:MANY, UPDATE( RESTRICT ),DELETE(RESTRICT )
                MANY:ONE          CASCADE          CASCADE
                CLEAR              CLEAR              CLEAR

filename FILE( {key} )
filename RELATED_FILE( {key} )
{FILE_TO_RELATED_KEY
  FIELD( filefieldname,relatedfilefieldname )      repeatable
        NOLINK,          NOLINK
  END}
{RELATED_FILE_TO_KEY
  FIELD( relatedfilefieldname,filefieldname )      repeatable
        NOLINK,          NOLINK
  END}
END
```

FILE_TO_RELATED_KEY

Defines the links between the two files (optional—for a “one-way” or “lookup” relationship). Linked fields are compared, and those records with matching values are deemed related.

FIELD Identifies a pair of linked fields between the related files (required).

***filefieldname* | NOLINK**

filefieldname is the label of the field in the first file that is linked (compared) to a field in the related file. NOLINK indicates no comparison is made to the field in the related file.

***relatedfilefieldname* | NOLINK**

relatedfilefieldname is the label of the field in the related file that is linked (compared) to a field in the first file. NOLINK indicates no comparison is made to the field in the first file.

RELATED_FILE_TO_KEY

Defines the links between the two files (optional—for a “one-way” or “lookup” relationship). Linked fields are compared, and those records with matching values are deemed related.

FIELD

Identifies a pair of linked fields between the related files (required).

***relatedfilefieldname* | NOLINK**

relatedfilefieldname is the label of the field in the related file that is linked (compared) to a field in the first file. NOLINK indicates no comparison is made to the field in the first file.

***filefieldname* | NOLINK**

filefieldname is the label of the field in the first file that is linked (compared) to a field in the related file. NOLINK indicates no comparison is made to the field in the related file.

Example—Relation Definition Group

[RELATIONS]

```

OrderDetails
Products
    RELATION, MANY:ONE, UPDATE(RESTRICT), DELETE(RESTRICT)
    FILE(DTL:KeyProductNumber)
    RELATED_FILE(PRO:KeyProductNumber)
    FILE_TO_RELATED_KEY
        FIELD(DTL:ProductNumber, PRO:ProductNumber)
    END
    RELATED_FILE_TO_KEY
        FIELD(PRO:ProductNumber, DTL:ProductNumber)
    END
    END
Customers
States
    RELATION, MANY:ONE
    FILE()
    RELATED_FILE(STA:KeyStateCode)
    FILE_TO_RELATED_KEY
        FIELD(CUS:State, STA:StateCode)
    END
    END
Customer
Orders
    RELATION, ONE:MANY, UPDATE(CASCADE), DELETE(RESTRICT)
    FILE(CUS:KeyCustNumber)
    RELATED_FILE(ORD:KeyCustDate)
    FILE_TO_RELATED_KEY
        FIELD(CUS:CustNumber, ORD:CustNumber)
        FIELD(NOLINK, ORD:Date)
    END
    RELATED_FILE_TO_KEY
        FIELD(ORD:CustNumber, CUS:CustNumber)
    END
    END

```

Common Subsections

The following subsections appear in many instances within the .TXD file. Their syntax and structure is consistent wherever they are found within the file.

[LONGDESC]

Lists up to 1000 characters of descriptive text on multiple lines (optional). Each line of text begins with an exclamation point (!) and contains up to 75 characters. The text comes from the **Comments** tab of the respective **Properties** dialog for the Dictionary, File, Alias, Relation, Field, or Key. For example:

```
[LONGDESC]
!Up to 1000 characters of text describing this item.
!Each line of up to 75 characters begins with an exclamation point.
```

[USEROPTION]

Lists up to 1000 characters of text that is available to templates (optional). Each line of text begins with an exclamation point (!) and contains up to 75 characters.

The text is available to any templates that process the file, alias, field, key, etc. See the EXTRACT procedure documented in the *Template Language Reference* for more information.

The text comes from the **Options** tab of the respective **Properties** dialog for the File, Alias, Relation, Field, or Key. For example:

```
[USEROPTION]
!ThirdPartyTemplateAttribute(on)
!DefaultWindowSize=Max
```

[TOOLOPTION]

Lists up to 1000 characters of text that is available to templates (optional). Each line of text begins with an exclamation point (!) and contains up to 75 characters.

The text is available to any templates that process the file, alias, field, key, etc. See the EXTRACT procedure documented in the *Template Language Reference* for more information.

The text comes from third-party add-ons and does not appear on any dialog in the Dictionary Editor. For example:

```
[TOOLOPTION]
!MyTool('Global option = A')
!HisTool('True')
```

[SCREENCONTROLS]

Marks the beginning of the subsection that describes the default window controls used to manage a data dictionary field, MEMO, or BLOB (optional).

Following [SCREENCONTROLS], an exclamation point (!) plus a space marks the beginning of a control declaration. The control declaration is the Clarion language statement that defines a window control used to manage the data item. There may be several controls associated with the data item, so there may be several control declarations, beginning immediately after [SCREENCONTROLS] and continuing until the next subsection, usually [REPORTCONTROLS], begins. For example:

```
[SCREENCONTROLS]
! PROMPT('CurrentTab:'),USE(?CurrentTab:Prompt)
! ENTRY(@s80),USE(CurrentTab)
[REPORTCONTROLS]
```

[REPORTCONTROLS]

Marks the beginning of the subsection that describes the default report controls used to manage a data dictionary field, MEMO, or BLOB (optional).

Following [REPORTCONTROLS], an exclamation point (!) plus a space marks the beginning of a control declaration. The control declaration is the Clarion language statement that defines a report control used to manage the data item. There may be several controls associated with the data item, so there may be several control declarations, beginning immediately after [REPORTCONTROLS] and continuing until the field definition begins. For example:

```
[REPORTCONTROLS]
! STRING(@s80),USE(CurrentTab)
```


11 - .TXA FILE FORMAT

.TXA Files: Clarion Applications in ASCII Format

.TXA files are simply ASCII text versions of Clarion .APP files. .APP files are where the Clarion for Windows environment stores all the application specific information necessary to generate and make an application. A .TXA file contains all the information stored in an .APP file (except for project system information), but in a text format that is readable (and writable) by most text editors.

You can easily create a .TXA file for any .APP file by loading the application into the Clarion for Windows development environment, then choosing **File** ► **Export Text** from the menu.

Why would you need a .TXA file? For at least three reasons:

- | | |
|--------------|--|
| Backup | Because a .TXA file can be imported just as easily as it is exported, it may serve as a backup of your .APP file. |
| Mass Changes | Because a .TXA file may be manipulated in your favorite text editor, you can use the power of the text editor to make mass changes to your application, or for that matter, any changes that would be easier with a text editor. |
| First Aid | Occasionally, an .APP file may exhibit some strange behavior in the development environment. Exporting to a .TXA file, then importing from that same file can, in some cases, create a clean .APP file. |

.TXA File Organization

The organization and syntax of the .TXA file reflects the Application-Program-Module-Procedure paradigm on which Clarion for Windows generated applications are based. The Class Clarion templates define this paradigm.

There are five major sections in the .TXA file that roughly correspond to the files produced by the Application Generator.

- ◆ [APPLICATION] section corresponds to the *appname.APP* file.
- ◆ [PROJECT] section contains the project system settings within the *appname.APP* file.
- ◆ [PROGRAM] section corresponds to the *appname.CLW* file.
- ◆ [MODULE] sections correspond to the *appna00n.CLW* files.
- ◆ [PROCEDURE] sections correspond to the procedures defined in the Application Tree and stored in the *appna00n.CLW* files.

.TXA Skeleton

On the following page is an ordered list of .TXA sections, subsections, and keywords. This list is designed to give you a feel for the overall structure and organization of the .TXA file.

Each section begins with its title enclosed in square brackets and ends with [END], or with the beginning of another section. Each section may contain subsections and keywords.

Subsections begin just like the major sections—with a title surrounded by square brackets.

Keywords appear in all capitals followed by the keyword value. The keyword values appear in various formats described below on a case by case basis.

The indentations in the list are provided here for readability and do not appear in the actual .TXA file.

Following the skeleton is a detailed discussion of each .TXA section, subsection, and keyword.

[APPLICATION]			
VERSION	optional		
HLP	optional		
DICTIONARY	optional		
PROCEDURE	optional		
[COMMON]			
DESCRIPTION	optional		} Common Body
LONG	optional		
FROM			
[DATA]			
[FILES]	optional		
[PROMPTS]			
[EMBED]	optional		
[ADDITION]	optional	<u>repeatable</u>	
[PERSIST]			
[PROJECT]			
[PROGRAM]			
NAME	optional		
INCLUDE	optional		
NOPOPULATE	optional		
[COMMON]			
DESCRIPTION	optional		} Common Body
LONG	optional		
FROM			
[DATA]			
[FILES]	optional		
[PROMPTS]			
[EMBED]	optional		
[ADDITION]	optional	<u>repeatable</u>	
[PROCEDURE]	optional	<u>repeatable</u>	
NAME	optional		
PROTOTYPE	optional		
[COMMON]			
DESCRIPTION	optional		} Common Body
LONG	optional		
READONLY	optional	procedure only	
FROM			
[DATA]			
[FILES]	optional		
[PROMPTS]			
[EMBED]	optional		
[ADDITION]	optional	<u>repeatable</u>	
[CALLS]	optional		
[WINDOW]	optional		
[REPORT]	optional		
[FORMULA]	optional		
[END]			
[MODULE]	optional	<u>repeatable</u>	
NAME	optional		
INCLUDE	optional		
NOPOPULATE	optional		
[COMMON]			
DESCRIPTION	optional		} Common Body
LONG	optional		
FROM			
[DATA]			
[FILES]	optional		
[PROMPTS]			
[EMBED]	optional		
[ADDITION]	optional	<u>repeatable</u>	
[PROCEDURE]	optional	<u>repeatable</u>	
.			
.			
[END]			

.TXA File Sections

[APPLICATION]

The application section is required and appears only once at the top of each .TXA file (unless only a portion of the application is exported, e.g. a .TXA file may contain a single module or procedure, in which case the file begins with [MODULE] or [PROCEDURE] respectively). This section begins with [APPLICATION] and ends with the beginning of the [PROGRAM] section. The application section contains the following keywords and subsections that pertain to the entire application.

```
[APPLICATION]
  VERSION          optional
  HLP              optional
  DICTIONARY       optional
  PROCEDURE        optional
  [COMMON]
    DESCRIPTION    optional
    LONG           optional
    FROM
  [DATA]
  [FILES]          optional
  [PROMPTS]
  [EMBED]          optional
  [ADDITION]       optional    repeatable
  [PERSIST]
```

VERSION The version number of the application (optional). This value reflects the version of generator, and changes when the TXA format changes. For example:

```
VERSION 10.
```

HLP The windows help file called by the application (optional). The file name may be fully qualified or not. If not, Clarion searches in the current directory, the system path, then in paths specified by the redirection file (C:\CW15\BIN\CW15.RED). For example:

```
HLP 'C:\CW15\APPS\MY.APP.HLP'
```

DICTIONARY The data dictionary file used by the application (optional). The file name may be fully qualified or not. If not, Clarion searches the paths specified by the redirection file (C:\CW15\BIN\CW15.RED). For example:

```
DICTIONARY 'TUTORIAL.DCT'
```

PROCEDURE	The name of the first procedure in the application (optional—no meaning for a .LIB or .DLL). This procedure calls all other procedures in the application, either directly, or indirectly. For example: PROCEDURE Main
[COMMON]	The common subsection appears in the [APPLICATION], [PROGRAM], [MODULE], and [PROCEDURE] sections (required). This subsection begins with [COMMON] and ends with the beginning of the next subsection. This subsection can vary substantially in length and appearance, depending on the section in which it resides and on the subsections it contains or omits. See the [COMMON] section below for a full discussion.
[PERSIST]	Information about the application that is “remembered” across sessions (required). Although these items appear in .TXA [PROMPT] format, they do not appear to the developer as prompts, rather, they are #DECLARED in the application template with the SAVE attribute, which causes them to be saved in the .APP file so they are available for each new session.

See [PROMPT] below for more information on the syntax of these application keywords.

Example—[APPLICATION]

```
[APPLICATION]
VERSION 10
HLP 'C:\CW15\APPS\MY.APP.HLP'
DICTIONARY 'TUTORIAL.DCT'
PROCEDURE Main
[COMMON]
.
.
.
```

[PROJECT]

The project section is always present and appears only once, after the [APPLICATION] section of each .TXA file. The project section begins with [PROJECT] and ends with [PROGRAM]. It is generated (exported) automatically for each application.

This section contains the project system settings specified for this application. It is provided so that project system settings are preserved when you export an application to a .TXA file, then import the .TXA back into an .APP file.

```
[PROJECT]
-- Generator
#noedit
#system win
#model clarion dll
#pragma debug(vid=>full)
#compile BIG_RD.Clw /define(GENERATED=>on)-- GENERATED
#compile BIG_RU.Clw /define(GENERATED=>on)-- GENERATED
#compile BIG_SF.Clw /define(GENERATED=>on)-- GENERATED
#compile ResCode.Clw /define(GENERATED=>on)-- GENERATED
#compile BIG.clw /define(GENERATED=>on)-- GENERATED
#compile BIG001.clw /define(GENERATED=>on)-- GENERATED
#compile BIG002.clw /define(GENERATED=>on)-- GENERATED
#pragma link(C%L%2AS4%S%.LIB)-- GENERATED
#link BIG.EXE
[PROGRAM]
```

See *TopSpeed's Project System* for complete information on project system syntax and pragmas.

[PROGRAM]—[END]

The program section is required and appears only once, after the [APPLICATION] section of each .TXA file. The program section begins with [PROGRAM] and ends with [END]. This section contains the following keywords and subsections that pertain to the source file that contains the PROGRAM statement.

```
[PROGRAM]
  NAME                optional
  INCLUDE             optional
  NOPOPULATE         optional
  [COMMON]
    DESCRIPTION      optional
    LONG             optional
    FROM
  [DATA]
  [FILES]            optional
  [PROMPTS]
  [EMBED]            optional
  [ADDITION]         optional   repeatable
  [PROCEDURE]       optional   repeatable
[END]
```

NAME The name of the source file that contains the PROGRAM statement for this application (optional).). If omitted it defaults to the application name. For example:

```
NAME ' TUTORIAL.CLW'
```

INCLUDE The name of a source file that is included in the data declaration section of the program source file (optional). For example:

```
INCLUDE 'EQUATES.CLW'
```

NOPOPULATE The Application Generator may not store procedures in this source file (optional). This is typically present for external modules. For example:

```
NOPOPULATE
```

[COMMON] The common subsection appears in the [APPLICATION], [PROGRAM], [MODULE], and [PROCEDURE] sections (required). This subsection begins with [COMMON] and ends with the beginning of the next subsection. This subsection can vary substantially in length and appearance, depending on the section in which it resides and on the subsections it contains or omits. See the [COMMON] section below for a full discussion.

[PROCEDURE]

Information that fully defines a procedure (optional). The procedure subsection may be repeated for each procedure in the module. The information stored in this subsection applies only to the procedure identified by the NAME keyword.

See the *[PROCEDURE]* section below for more information on the structure and syntax of this subsection.

Example—[PROGRAM]-[END]

```
[PROGRAM]
NAME ' TUTORIAL.CLW'
INCLUDE 'EQUATES.CLW'
NOPOPULATE
[COMMON]
.
.
.
[END]
```

[MODULE]—[END]

The structure and syntax of the module section is identical to that of the program section. The module section is optional and may be repeated as many times as necessary. The section begins with [MODULE] and ends with [END].

Although identical in syntax and structure, the module subsection differs in the scope of its applicability. The module section is repeated once for each module in the application, and, the information it contains applies only to the source file identified by its NAME keyword, that is, only to those procedures that reside within this module. Data defined in its [DATA] section is “module” data, and is available to all procedures in the module.

[PROCEDURE]

The procedure subsection is optional and is repeated for each procedure in the program or module. The subsection begins with [PROCEDURE] and ends with the next [PROCEDURE], or the [END] of the module or program. This subsection contains information that pertains only to the procedure identified by the NAME keyword.

```
[PROCEDURE]      optional      repeatable
NAME             optional
PROTOTYPE       optional
[COMMON]
DESCRIPTION     optional
LONG            optional
READONLY       optional
FROM
[DATA]
[FILES]         optional
[PROMPTS]
[EMBED]         optional
[ADDITION]      optional      repeatable
[CALLS]         optional
[WINDOW]       optional
[REPORT]       optional
[FORMULA]      optional
```

NAME The name of the procedure the keywords in this section apply to (optional). For example:

```
NAME BrowseCustomers
```

PROTOTYPE The prototype for the procedure (optional). See the *Language Reference* for more information on prototyping your procedures. For example:

```
PROTOTYPE 'LONG Count, REAL Sum'
```

[COMMON] The common subsection appears in the [APPLICATION], [PROGRAM], [MODULE], and [PROCEDURE] sections (required). This subsection begins with [COMMON] and ends with the beginning of the next subsection. This subsection can vary substantially in length and appearance, depending on the section in which it resides and on the subsections it contains or omits. See the [COMMON] section below for a full discussion.

[CALLS] Procedures called by this procedure (optional). For example:

```
[CALLS]
UpdateCustomers
BrowseOrders
```

[WINDOW] Clarion Language statements that define the window managed by this procedure (optional).

In addition to the Clarion Language statements defining the WINDOW structure, the window subsection may contain the following four keywords which are used internally by the development environment:

#SEQ(*instance number*)

All controls populated from a control template have this keyword which gives the instance number of the control template of which they are a member.

#ORIG(*original name of control*)

The original name of the control as given in the control template.

#FIELDS(*list of fields in a list box*)

The list of fields to display in a LIST control.

#LINK(*field equate label of linked fields*)

If several controls are populated from the same control template, they are linked together in a cycle, each being linked to the next. If a field is populated from the dictionary the prompts are also linked to the entry fields.

For example:

```
[WINDOW]
QuickWindow WINDOW('Browse the Customers File'),AT(,358,188),SYSTEM,GRAY,MDI
LIST,AT(8,20,342,124),USE(?Browse:1),IMM,HVSCROLL,FORMAT('16L|M~Cust' &|
'Number~@n4@80L|M~Company Name~@S20@80L|M~Address~@S20@80L|M~City~@S20' &|
'@8L|M~State~@S2@20L|M~Zip~@S5@'),FROM(Queue:Browse:1),#SEQ(1),#ORIG(?List),|
#FIELDS(CUS:CustNumber,CUS:CompanyName,CUS:Address,CUS:City,CUS:State,CUS:Zip)
BUTTON('&Insert'),AT(207,148,45,14),USE(?Insert:2),#SEQ(2),#ORIG(?Insert),#LINK(?Change:2)
BUTTON('&Change'),AT(256,148,45,14),USE(?Change:2),DEFAULT,#SEQ(2),#ORIG(?Change),#LINK(?Delete:2)
BUTTON('&Delete'),AT(305,148,45,14),USE(?Delete:2),#SEQ(2),#ORIG(?Delete),#LINK(?Insert:2)
END
```

[REPORT] Clarion Language statements that define the REPORT managed by this procedure (optional). In addition to the Clarion Language statements, the report subsection may contain the four keywords above.

[FORMULA]

Describes each formula defined for this procedure by the **Formula Editor** (optional). Notice that the keywords and their values correspond exactly to the **Formula Editor** dialog. See the *User's Guide* and the on-line help for more information on these fields. For example:

```
[FORMULA]
DEFINE OrderTax
ASSIGN OrderTax
CLASS After Lookups
DESCRIPTION Calculate State Tax
= OrderTotal * StateTaxRate
[END]
```

Example—[PROCEDURE]

```
[PROCEDURE]
NAME BrowseCustomers
PROTOTYPE 'LONG Count, REAL Sum'[CALLS]
[COMMON]
.
.
.
[CALLS]
UpdateCustomers
BrowseOrders
[WINDOW]
QuickWindow WINDOW('Browse Customers'),AT(,358,188),SYSTEM,GRAY,MDI
LIST,AT(8,20,342,124),USE(?Browse:1),IMM,FORMAT('16L|M~Cust' &|
'Number~@n4@80L|M~Name~@S20@80L|M~Address~@S20@80L|M~City~@S20' &|
'@8L|M~St~@S2@20L|M~Zip~@S5@'),FROM(Queue:Browse:1),#SEQ(1),#ORIG(?List),|
#FIELDS(CUS:CustNumber,CUS:CompanyName,CUS:Address,CUS:City,CUS:State,CUS:Zip)
BUTTON('&Insert'),AT(207,148,45,14),USE(?Insert:2),#SEQ(2),#ORIG(?Insert),#LINK(?Change:2)
BUTTON('&Change'),AT(256,148,45,14),USE(?Change:2),DEFAULT,#SEQ(2),#ORIG(?Change),#LINK(?Delete:2)
BUTTON('&Delete'),AT(305,148,45,14),USE(?Delete:2),#SEQ(2),#ORIG(?Delete),#LINK(?Insert:2)
END
[FORMULA]
DEFINE OrderTax
ASSIGN OrderTax
CLASS After Lookups
DESCRIPTION Calculate State Tax
= OrderTotal * StateTaxRate
```

Common Subsections

[COMMON]

The common subsection appears in the [APPLICATION], [PROGRAM], [MODULE], and [PROCEDURE] sections. This subsection begins with [COMMON] and ends with the beginning of the next subsection. This subsection is required, although many of its keywords and subsections are optional. This subsection can vary substantially in length and appearance, depending on the section in which it resides and on the subsections it includes or omits. [COMMON] contains the following keywords and subsections.

[COMMON]		
DESCRIPTION	optional	
LONG	optional	
FROM		
MODIFIED		
[DATA]		
[FILES]	optional	
[PROMPTS]		
[EMBED]	optional	
[ADDITION]	optional	repeatable

DESCRIPTION Up to 40 characters of text describing the application, program, module, or procedure (optional). For example:

```
DESCRIPTION 'Print dynamic label report'
```

LONG Up to 1000 characters of text describing the application, program, module, or procedure (optional). For example: The text is actually split into several lines that are concatenated together as they are read. For example:

```
LONG 'Print dynamic label report. At runtime, the '
LONG 'user selects (or adds a new description of) '
LONG 'a label paper from the LAB file. This proced'
LONG 'ure then makes property assignments to adjus'
LONG 't the size and location of the label text.'
```

READONLY The procedure may be viewed, but not modified from the Clarion for Windows environment (optional). Only allowed in a [PROCEDURE] subsection. READONLY cannot currently be added to a procedure by the environment, but is provided for future use so that TopSpeed's developers can implement multi-developer environments that allow a procedure to be "checked out" and "checked in" in order to preserve code integrity. For example:

```
READONLY
```

FROM The name of the template class for an application, or the name of the template class and the specific template from which the program, module, or procedure is generated (optional - can only be omitted for a ToDo procedure). For example:

```
[APPLICATION]
```

```
.  
.  
.
```

```
FROM Clarion
```

or

```
[PROCEDURE]
```

```
.  
.  
.
```

```
FROM Clarion Report
```

MODIFIED The date and time the procedure was last modified. For example:

```
[PROCEDURE]
```

```
MODIFIED '1998/07/02' '10:43:32'
```

Example—[COMMON]

```
[COMMON]
```

```
DESCRIPTION 'Print dynamic label report'
```

```
LONG 'Print dynamic label report. At runtime, the user selects (or adds a  
new description of) a label paper from the LAB file. This procedure then  
makes property assignments to adjust the size and location of the label  
text to fit the selected label paper.'
```

```
READONLY
```

```
FROM Clarion Report
```

[DATA]

The data subsection is an optional part of the [COMMON] subsection. It may contain several subsections and keywords that describe each memory variable defined for this procedure, module, program, or application. See the *.TXD File Format* chapter of this book for a discussion of how this same syntax applies to data dictionary fields. Also see *Defining Procedure Data* in the *User's Guide*.

For each memory variable defined, there is a series of *optional* subsections and keywords that fully describe the variable, as well as any default formatting conventions the Application Generator is expected to follow. The complete list of possible subsections and keywords is:

[DATA]	
[LONGDESC]	optional
[USEROPTION]	optional
[SCREENCONTROLS]	optional
[REPORTCONTROLS]	optional
field definition	
keyword list	optional

[LONGDESC] Up to thirteen (13) lines of text, up to seventy-five (75) characters in length each (optional). Each line of text begins with an exclamation point (!). Comes from the **Comments** tab of the **Field Properties** dialog. For example:

```
[LONGDESC]
!CurrentTab is used internally by the template
!generated code to store the number/id of the
!TAB control that has focus.
```

[USEROPTION] Up to thirteen (13) lines of text up to seventy-five (75) characters in length each (optional). Each line of text begins with an exclamation point (!). The text is available to templates. See the Extract procedure in the *Template Language Reference*. Comes from the **Options** tab of the **Field Properties** dialog. For example:

```
[USEROPTION]
!ThirdPartyTemplateAttribute:Details(on)
!ThirdPartyTemplateAttribute:WizardHelp(off)
```

[SCREENCONTROLS] The beginning of the subsection that describes the default window controls used to manage the memory variable (optional). Use the **Window** tab of the **Field Properties** dialog to set the default controls.

Following **[SCREENCONTROLS]**, an exclamation point (!) marks the beginning of a control declaration. The control declaration is the Clarion language statement that defines a window control. There may be several controls associated with the memory variable, so there may be several control declarations, beginning immediately after **[SCREENCONTROLS]** and continuing until the next subsection, usually **[REPORTCONTROLS]**. For example:

```
[SCREENCONTROLS]
! PROMPT('CurrentTab:'),USE(?CurrentTab:Prompt)
! ENTRY(@s80),USE(CurrentTab)
```

[REPORTCONTROLS]

The beginning of the subsection that describes the default report controls used to manage the memory variable (optional). Use the **Report** tab of the **Field Properties** dialog to set the default controls.

Following **[REPORTCONTROLS]**, an exclamation point (!) marks the beginning of a control declaration. The control declaration is the Clarion language statement that defines a report control. There may be several controls associated with the memory variable, so there may be several control declarations, beginning immediately after **[REPORTCONTROLS]** and continuing until the field definition begins. For example:

```
[REPORTCONTROLS]
! STRING(@s80),USE(CurrentTab)
```

Field Definition

Lists the Clarion language field declaration (required). Optionally includes a text description of up to 40 characters. The text description begins with an exclamation point (!). For example:

```
CurrentTab  STRING(80) !user selected tab
```

Keyword List

The keywords that specify various attributes of the memory variable (optional). The list begins with !!>.

The keywords in this list are set using the **Field Properties** dialog. Many of the keywords correspond directly to Clarion language keywords. See the *Language Reference* for more information on these keywords.

See the *.TXD File Format* chapter of this book for a complete discussion of the keywords and their effects. Also see the *User's Guide*, the *Language Reference*, and the on-line help for more information on particular Clarion language keywords.

Notice that the keywords in the **[DATA]** subsection correspond closely to the tabs and prompts in the **Field Properties** dialog. This is because the **Field Properties** dialog is where these values are set

Example—[DATA]

For any given variable, you will usually see only a fraction of the possible subsections and keywords, because, some are mutually exclusive, and many others are simply not required. The one item that is *required* is the Clarion Language field definition.

Let's examine each line of the following typical example to illustrate how this subsection works:

```
[DATA]
[SCREENCONTROLS]
! PROMPT('CurrentTab:'),USE(?CurrentTab:Prompt)
! ENTRY(@s80),USE(CurrentTab)
[REPORTCONTROLS]
! STRING(@s80),USE(CurrentTab)
CurrentTab          STRING(80) ! Tab selected by user
!!> IDENT(4294967206),PROMPT('CurrentTab:'),HEADER('CurrentTab'),
PICTURE(@s80)
[SCREENCONTROLS]
.
.
.
```

[DATA] marks the beginning of this subsection which describes all the memory variables for this application, program, module, or procedure.

[SCREENCONTROLS] marks the beginning of the subsection that describes the default window controls used to manage the first memory variable.

Following [SCREENCONTROLS], the exclamation point (!) marks the beginning of a control declaration. The control declaration is the Clarion language statement that defines a window control used to manage the memory variable. There may be several controls associated with the variable, so there may be several control declarations beginning immediately after [SCREENCONTROLS] and continuing until a new subsection, usually [REPORTCONTROLS], begins.

[REPORTCONTROLS] marks the end of the [SCREENCONTROLS] subsection, and the beginning of the subsection that describes the default report controls used to display the memory variable. The control declaration is the Clarion language statement that defines a report control used to manage the memory variable. There may be several controls associated with the variable, so there may be several control declarations beginning immediately after [REPORTCONTROLS] and continuing until the field definition begins.

Following the report control declaration, is the Clarion Language field definition for the variable: "CurrentTab String(80)." *This is the only required keyword* for each memory variable described in the .TXA file. It may optionally be followed by an exclamation point (!) and the short description for the variable.

Finally, "!!>" marks the beginning of the keyword list associated with the memory variable. The keywords are separated by commas, and the list continues, wrapping onto multiple lines if necessary, until the next subsection begins. See the *.TXD File Format* chapter of this book for a complete discussion of the keywords and their effects.

[FILES]

The files subsection is an optional part of the [COMMON] subsection. It may contain several subsections and keywords that describe the files used by this procedure, module, program, or application. In Class Clarion generated applications, the [FILES] subsection is most commonly seen in the [PROCEDURE] subsection, since the procedures do most of the file access, while applications, programs, and modules are more concerned with managing the user's environment.

For each file used by the procedure, module, program, or application, there is a series of subsections and keywords that identify the file, the key, and any related files that will also be used. The complete list of possible subsections and keywords is:

```
[FILES]                optional
  [PRIMARY]            optional, repeatable
    [INSTANCE]
      [KEY]             optional
      [SECONDARY]      optional, repeatable
    [OTHERS]           optional
      .
      .
      .
```

[FILES] The beginning of this subsection which identifies the essential information about the files used by this procedure (optional).

[PRIMARY] The beginning of the subsection which describes a single primary file tree for a control template in this procedure (required). There may be a [PRIMARY] subsection for each control template in the procedure.

Primary simply means that this is the main file processed by the associated control template. Any other files processed by the control template are dependent files.

The line immediately after [PRIMARY] lists the filename. For example:

```
[PRIMARY]
Customers
```

[INSTANCE] Introduces the instance number of the control template for which this file is primary (required). See *Common Subsections—[ADDITION]* for more information on instance numbers. For example:

```
[INSTANCE]
6
```

[KEY] Introduces the key used to access this file (optional). For example:

```
[KEY]
CUS:KeyCustNumber
```

[SECONDARY] Introduces the child and the parent file accessed by this control template (optional). This subsection is repeated for each related file.

Both files are named in order to avoid any ambiguity when there is more than one related file. The first file listed is always the “child” file, and the second file listed is always the “parent” file.

The [SECONDARY] file identities are part of the [PRIMARY] subsection, that is, a [SECONDARY] does not mark the end of the [PRIMARY] subsection but is a continuation of it. For example:

```
[SECONDARY]
Phones Customers
```

[OTHERS] Introduces other files accessed by the procedure, but not by a control template (optional). It also marks the end of the prior [PRIMARY] subsection. The only code generated for an [OTHERS] file is just that code necessary to open the file at the beginning of the procedure, then close the file at the end of the procedure. For example:

```
[OTHERS]
Labels
```

Example—[FILES]

```
[FILES]
[PRIMARY]
Customers
[INSTANCE]
6
[KEY]
CUS:KeyCustNumber
[SECONDARY]
Phones Customers
[SECONDARY]
Orders Customers
[OTHERS]
Labels
.
.
.
```

Let's examine each line of the following comprehensive example to illustrate how this subsection works:

[FILES] marks the beginning of this subsection which identifies the essential information about the files used by this procedure.

[PRIMARY] marks the beginning of the subsection which describes a single primary file tree for a control template. Primary simply means that this is the main file processed by the associated control template. Any other files processed by the control template are dependent files.

The line immediately after [PRIMARY] shows the filename, that is, Customers.

[INSTANCE] means the following line shows the instance number of the control template for which this file is primary. The development environment uses the instance number to link the appropriate file to the appropriate control template.

[KEY] means the following line shows the key used to access the primary, that is, CUS:KeyCustNumber.

[SECONDARY] means the following line shows a related file that is also accessed by this control template: Phones. Notice that the line naming the secondary file also names the parent file: Customers. This is to avoid any ambiguity when there is more than one related file. For example, if the Orders file was listed as a secondary file as well as the Phones file, it would be important to know if the Orders file is related to the Phones file or the Customers file.

Similarly, the order in which the two file names appear is significant. The first file listed is always the “child” file, and the second file listed is always the “parent” file.

Finally, [OTHERS] means the following lines show other files accessed by the procedure, but not a control template. It also marks the end of the prior [PRIMARY] subsection.

[PROMPTS]

The prompts subsection is part of the common subsection. It lists the template prompts associated with the application, program, module, or procedure, plus the values supplied for the prompts by the developer. See the *Template Language Reference* in the on-line help for more information on template prompts.

Template prompts may be found in several different subsections, including [PERSIST] (see [APPLICATION] above), [PROMPTS], and [FIELDPROMPT] (see [ADDITION] below).

The template prompts and their associated values appear in two different formats: simple and dependent.

Simple Prompts

Both formats begin with the prompt name. The prompt name begins with the percent sign (%). The **simple format** follows the prompt name with a prompt type and a value enclosed in parentheses.

Available prompt types are @picture, LONG, REAL, STRING, FILE, FIELD, KEY, COMPONENT, PROCEDURE, and DEFAULT. The prompt type may optionally be further qualified as UNIQUE or as MULTI. MULTI means the prompt has multiple values. UNIQUE also indicates multiple values, however, the values are in ascending order and there are no duplicates.

The simple format syntax is diagrammed as follows, where the vertical columns show alternative parameters, the curly braces show optional parameters, and *value* is the value for the prompt:

%name	{UNIQUE}	@picture	('value')
	{MULTI}	LONG	('value ₁ ','value ₂ ','value _n ')
		REAL	
		STRING	
		FILE	
		FIELD	
		KEY	
		COMPONENT	
		PROCEDURE	
		DEFAULT	

Following is a description of the available prompt types:

@picture	The value is a picture token.
LONG	The value is a number in LONG format (4 byte unsigned integer).

REAL	The value is a number in REAL format (8 byte floating point format).
STRING	The value is a character string.
FILE	The value is the label of a data file.
FIELD	The value is the label of a field in a data file.
KEY	The value is the label of a key.
COMPONENT	The value is the label of a key component.
PROCEDURE	The value is the label of a procedure.
DEFAULT	Typically the same as STRING, but stored differently internally. DEFAULT variables do not have an explicit type. If they are assigned to, the variable takes the type of the value that is assigned to it.

For example:

```
[PROMPTS]
%LastProgramExtension DEFAULT ('EXE')
%SizePreferences MULTI LONG (3,3)
%RangeField COMPONENT (PHO:CustNumber)
%UpdateProcedure PROCEDURE (UpdatePhones)
```

Let's examine each line of this example.

%LastProgramExtension is the template name of the stored value. DEFAULT indicates the value is stored as a string (since 'EXE' is a string). The ('EXE') indicates the value of %LastProgramExtension is 'EXE.'

%SizePreferences is the template name of the stored value. The value is stored in LONG format and MULTI means there are multiple values. The (3,3) indicates the values for %SizePreferences are 3 and 3.

%RangeField is the template name of the stored value. The value stored is the label of a KEY COMPONENT, and PHO:CustNumber is that label.

%UpdateProcedure is the template name of the information stored. The information stored is the label of a PROCEDURE and UpdatePhones is that label.

Dependent Prompts

Like simple prompts, dependent prompts begin with the prompt name. The prompt name begins with the percent sign (%). The **dependent format**

follows the prompt name with the keyword `DEPEND` and is spread over multiple lines.

The dependent format is diagrammed as follows, where the vertical columns show alternative parameters, the curly braces show optional parameters, and the `WHEN` lines represent all the possible values for both the `%Prompt` and the `%ParentSymbol`:

```
%Prompt   DEPEND  %ParentSymbol {UNIQUE} @picture TIMES n
                                     {MULTI} LONG
                                             REAL
                                             STRING
                                             FILE
                                             FIELD
                                             KEY
                                             COMPONENT
                                             DEFAULT
WHEN      ('ParentSymbolValue') ('promptvalue')1
WHEN      ('ParentSymbolValue') ('promptvalue')2
WHEN      ('ParentSymbolValue') ('promptvalue')n
```

The `%ParentSymbol` following the `DEPEND` keyword represents a template symbol that the value of `%Prompt` depends upon. That is, the `%ParentSymbol` may have multiple different values, and the value of `%Prompt` depends on the current value of `%ParentSymbol`.

For example:

```
[PROMPTS]
%GenerationCompleted DEPEND %Module DEFAULT TIMES 4
WHEN ('TUTORIAL.clw') ('1')
WHEN ('TUTOR001.clw') ('1')
WHEN ('TUTOR002.clw') ('1')
WHEN ('TUTOR003.clw') ('1')
.
.
.
```

Again, let's examine each line of the example. `%GenerationCompleted` is the name of the prompt for which the value is stored. The value of `%GenerationCompleted` `DEPENDS` on the value of `%Module`. The values are stored as strings. `TIMES 4` means `%Module` has 4 different possible values.

On the following 4 lines, 1 for each possible value of `%Module`, `WHEN` indicates a possible value for `%Module`—'TUTORIAL.CLW', followed by the corresponding value for `%GenerationCompleted`—'1'.

Nested Dependent Prompts

Dependent prompts can also show more than one level of dependency. The `WHEN` instances are nested for each additional level of dependency. For example:

```
%ForegroundNormal DEPEND %Control DEPEND %ControlField LONG TIMES 2  
WHEN ('?Browse:1') TIMES 2  
WHEN ('CUS:CustNumber') (4294967295)  
WHEN ('CUS:CompanyName') (4294967295)  
WHEN ('?Browse:2') TIMES 4  
WHEN ('CUS:Address') (4294967295)  
WHEN ('CUS:City') (4294967295)  
WHEN ('CUS:State') (4294967295)  
WHEN ('CUS:ZipCode') (4294967295)
```

In this example, the value of %ForegroundNormal depends on the value of %Control, and then on the value of %ControlField. %Control can have 2 possible values: ?Browse:1 and ?Browse:2. For each of these values, there is a WHEN...TIMES line that shows the number of possible values of %ControlField associated with this value of %Control.

Then, following each WHEN...TIMES line, are more WHEN lines showing each possible value for %ControlField, followed by the corresponding value for %ForegroundNormal. Note the precedence, %controlfield is dependent on the current value of %control.

[EMBED]—[END]

The embed subsection is an optional part of the common subsection. It may contain several subsections and keywords that describe each embed point defined for this procedure, module, or program with the **Embedded Source** dialog. See *Defining Embedded Source* in the *User's Guide*.

The [EMBED] subsection may contain the following subsections and keywords:

```

[EMBED]                optional
  EMBED                repeatable
    [INSTANCES]        optional, repeatable
      WHEN
        [DEFINITION]
          [SOURCE]      optional, repeatable
          [TEMPLATE]    optional, repeatable
          [PROCEDURE]   optional, repeatable
          [GROUP]       optional, repeatable
            INSTANCE 4
          [END]
        [END]
      [END]
    [END]
  [END]

```

[EMBED]-[END] Marks the beginning of this subsection that describes each embed point defined for this procedure (optional).

EMBED The string following this keyword identifies the embed point (required). EMBED appears once for each “filled” embed point. For example:

```
EMBED %ControlPreEventHandling
```

[INSTANCES]-[END]

Indicates there is more than one instance of this embed point (optional). See the example below.

WHEN

Indicates in which instance of the embed point the source statements are embedded (required). For example:

```

[INSTANCES]
WHEN '?Change:2'

```

[DEFINITION]-[END]

Marks the beginning of the subsection that defines the embedded source statements (required).

[SOURCE]	Identifies the source statements as free-form text from the Text Editor (optional). This may contain omissible information delimited by PROPERTY:BEGIN and PROPERTY:END statements. For example:
	<pre>[SOURCE] PROPERTY:BEGIN PRIORITY 4000 PROPERTY:END !This is a source embed point</pre>
[TEMPLATE]	Identifies the source statements as free-form text from the Text Editor which includes Template Language statements (optional). For example:
	<pre>[TEMPLATE] !This is a template coded embed point #FOR(%LocalData) ! %LocalData #ENDFOR</pre>
[PROCEDURE]	Identifies the embedded source statements as a procedure call from the Procedure to Call dialog. For example:
	<pre>[PROCEDURE] EmbeddedProcedureCall</pre>
[GROUP]	Identifies the embedded source as a code template.
INSTANCE	Identifies the instance number of the embedded code template. See <i>[ADDITION]</i> below. For example:
	<pre>[GROUP] INSTANCE 4</pre>

Example—[EMBED]-[END]

Let's examine a comprehensive example to illustrate each component of the **[EMBED]** subsection. First, notice how the .TXA text is very similar to the text in the **Embedded Source** dialog:

```
[EMBED]
EMBED %ControlPreEventHandling
[INSTANCES]
WHEN '?Change:2'
[INSTANCES]
WHEN 'Accepted'
[DEFINITION]
[SOURCE]
!This is a source embed point
[PROCEDURE]
EmbeddedProcedureCall
```

```

[GROUP]
INSTANCE 4
[END]
[END]
[END]
EMBED %ControlEventHandling
. . .
[END]

```

[EMBED] marks the beginning of the subsection. The [EMBED] subsection always ends with [END].

EMBED %ControlPreEventHandling identifies the embed point where the source is embedded. Notice that the name for the embed point in the [EMBED] subsection is slightly different than in the **Embedded Source** dialog. The names in the **Embedded Source** dialog are expanded for maximum clarity.

[INSTANCES] indicates there is more than one instance of the %ControlPreEventHandling embed point, that is, there is one instance of this embed point for each control in the procedure. Each [INSTANCES] ends with [END].

WHEN ‘?Change:2’ indicates in which instance of the embed point the source statements are embedded. At code generation time, there is a variable (or macro) named %Control that has the value ‘?Change:2’ indicating which control, and thus which instance of the embed point, is processed.

The fifth and sixth lines indicate yet another level of instances for this embed point. Not only does the embed point have an instance for each control in the procedure, it has an instance for each event associated with each control. At code generation time, there is a variable (or macro) named %ControlEvent that has the value ‘Accepted’ indicating which control event, and thus which instance of the embed point, is processed.

[DEFINITION] marks the beginning of the subsection that defines the embedded source statements. The [DEFINITION] ends with [END].

[SOURCE] indicates the type of embedded source: free-form source from the Text Editor. The free-form source follows on the next line and continues until the next .TXA subsection begins.

[PROCEDURE] indicates another type of embedded source: a procedure call. The procedure call follows on the next line.

[GROUP] indicates the third type of embedded source: a code template. Code templates are described in the [ADDITION] subsection. INSTANCE 4 indicates the instance subsection within the addition subsection where the embedded code template is described (see [ADDITION] below). (The name [GROUP], rather than [CODE] is used for historical reasons.)

[ADDITION]

The addition subsection is an optional part of the common subsection and appears once for each code template used. It may contain several subsections and keywords that describe each control, code, and extension template defined for this procedure, module, or program. See *Using Control, Code, and Extension Templates* in the *User's Guide*.

The [ADDITION] subsection may contain the following subsections and keywords:

[ADDITION]	repeatable
NAME	
[FIELDPROMPT]	optional
[INSTANCE]	repeatable
INSTANCE	
PARENT	optional
PROCPROP	optional
[PROMPTS]	optional

[ADDITION] Marks the beginning of the subsection (optional). Appears once for each template *type* used. That is, if BrowseBox is used twice in a procedure, there is only one BrowseBox [ADDITION] in the [PROCEDURE] section, but with multiple [INSTANCE]s (see below).

NAME Identifies the template class and the specific template invoked (required). Appears once for each [ADDITION] subsection. For example:

```
NAME Clarion BrowseUpdateButtons
```

[FIELDPROMPT] Indicates a prompt and its associated type and value (optional). This is only generated if you use a #FIELD statement in your templates. The prompts begin on the following line. See [PROMPTS] above for a full discussion of prompt syntax. For example:

```
[FIELDPROMPT]
%MadeItUp LONG (1)
```

[INSTANCE] Introduces the INSTANCE number on the following line. Appears once for each control, code, or extension template in the application, program, module, or procedure. See NAME above.

INSTANCE Indicates the instance number (identification number) of this particular template addition. For example:

```
[INSTANCE]
INSTANCE 2
```

PARENT	Indicates this control template depends on another control template (optional). PARENT is followed by the instance number of the control template upon which this control template depends. For example: PARENT 1
PROCPROP	Means the prompts for this control template are shown in the Procedure Properties dialog (optional). If PROCPROP is absent, the prompts will not be displayed in the Procedure Properties dialog. For example: PROCPROP
[PROMPTS]	Marks the beginning of the list of prompts for this control template, and the values supplied by the developer for each prompt (optional). The prompts begin on the following line and continue until the beginning of the next .TXA subsection. See the <i>[PROMPTS]</i> section above for a complete discussion of prompt syntax. For example: [PROMPTS] %UpdateProcedure PROCEDURE (UpdatePhones) %EditViaPopup LONG (1)

Example—[ADDITION]

Again, let's examine a comprehensive example to illustrate each component of the [ADDITION] subsection.

```
[ADDITION]
NAME Clarion BrowseUpdateButtons
[FIELDPROMPT]
%MadeItUp LONG (1)
[INSTANCE]
INSTANCE 2
PARENT 1
PROCPROP
[PROMPTS]
%UpdateProcedure PROCEDURE (UpdatePhones)
%EditViaPopup LONG (1)
.
.
.
[INSTANCE]
INSTANCE 4
PARENT 3
.
.
.
```

[ADDITION] marks the beginning of the subsection.

NAME identifies the template class (Clarion) and the specific template (BrowseUpdateButtons) invoked.

[FIELDPROMPT] indicates a prompt and its associated type and value. This is only generated if you use a #FIELD statement in your templates. The prompts begin on the following line. See [PROMPTS] above for a full discussion of prompt syntax.

[INSTANCE] introduces the INSTANCE number. On the following line, INSTANCE 2 indicates the instance number of this particular template addition.

PARENT 1 indicates this control template depends on another control template whose INSTANCE number is 1. That is, the BrowseUpdateButtons template only makes sense if there is also a BrowseBox template. Further, the BrowseUpdateButtons are associated with this *particular* BrowseBox whose INSTANCE number is 1. This is very important when there is more than one BrowseBox in the procedure.

PROCPROP means the prompts for this control template are shown in the **Procedure Properties** dialog. If PROCPROP is absent, the prompts will not be displayed in the **Procedure Properties** dialog.

[PROMPTS] marks the beginning of the list of prompts for this control template, and the values supplied by the developer for each prompt. The prompts begin on the following line and continue until the beginning of the next .TXA subsection. See the [PROMPTS] section above for a complete discussion of this subsection.

12 - TOPSPEED'S PROJECT SYSTEM

Introduction

The TopSpeed Project System, is integrated into the Clarion Environment. It is a powerful sequential language which combines the functionality of a batch processor, a linker and an intelligent compile-and-link system.

The TopSpeed Project System gives you total control over the compile and link process for the simplest single .EXE project up to the most complicated multiple .DLL project.

The primary benefits of using the TopSpeed Project System are automation, efficiency, and accuracy. With a single command, you can remake your entire project, no matter how complicated, and you can be assured that the correct source and objects are included in the compile and link processes, plus, the components that don't need it, don't get reprocessed. In addition, you can make different versions of your project (release version, debug version, evaluation/demo version, etc.) with the flip of a switch.

Here is a simple example of some project system language generated by the Clarion Application Generator:

```
#noedit
#system win
#model clarion dll
#pragma debug(vid=>full)
#compile QWKTU_RD.CLW- GENERATED
#compile QWKTU_RU.CLW- GENERATED
#compile QWKTU_SF.CLW- GENERATED
#compile QWKTUTOR.clw /define(GENERATED=>on)- GENERATED
#compile QWKTU001.clw /define(GENERATED=>on)- GENERATED
#compile QWKTU002.clw /define(GENERATED=>on)- GENERATED
#compile QWKTU003.clw /define(GENERATED=>on)- GENERATED
#pragma link(C%L%TPS%S%.LIB)- GENERATED
#link QWKTUTOR.EXE
```

Language Components

Keywords start with a pound sign (#). In the example, each keyword begins on a new line for readability. This is not required.

Comments start with a double hyphen (--) and are terminated by a Carriage Return or Line Feed.

Macros are surrounded by percent signs (%). You may want to think of macros as variables—a value is substituted whenever the project system

encounters a macro name surrounded by percent signs (%). See *Project System Macros* below.

Keyword Parameters are everything else you see in the example. Parameters and their syntax are discussed with each keyword.

The following keywords are recognized by the Project System:

<u>#abort</u>	<u>#expand</u>	<u>#older</u>
<u>#and</u>	<u>#file</u>	<u>#or</u>
<u>#autocompile</u>	<u>#if</u>	<u>#pragma</u>
<u>#compile</u>	<u>#ignore</u>	<u>#prompt</u>
<u>#declare_compiler</u>	<u>#implib</u>	<u>#run</u>
<u>#dmlink</u>	<u>#include</u>	<u>#rundll</u>
<u>#else</u>	<u>#link</u>	<u>#set</u>
<u>#elseif</u>	<u>#message</u>	<u>#split</u>
<u>#endif</u>	<u>#model</u>	<u>#system</u>
<u>#error</u>	<u>#noedit</u>	<u>#then</u>
<u>#exemod</u>	<u>#not</u>	<u>#to</u>
<u>#exists</u>		

Files and Editing

With regard to Clarion, the project system commands are generally stored in either a .PRJ file or an .APP file. APP files are maintained strictly through Clarion's development environment, however, PRJ files are simple ASCII text and may be maintained with the development environment (See the *User's Guide; Using the Project System*) or with your favorite text editor.

#noedit

The #noedit command can be placed at the top of a project file to prevent menu-editing from the TopSpeed environment. It has no effect in the Clarion environment.

Project System Macros

Macros are special strings that indicate a variable substitution is required. You may find it useful to think of macros as variables.

A sequence of characters enclosed by % characters indicates a macro name. The following characters are permitted in macro names:

```
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m
n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9 _
```

The trailing % may be omitted provided the character following the macro name is *not* one of the characters above.

Whenever a % delimited macro name is encountered, it is replaced either by the string associated with that macro, or by an empty string if there is no associated string. Substitution strings are associated with a macro by using the #set command.

Two adjacent % characters may be used when a % character is required in the substituted string. This double % technique can be used to delay macro substitution. For example:

```
#set echo = '#message %mymac'--'#message %mymac' associated with echo

#set mymac = 'Hello'          --'Hello' associated with mymac

#echo                        --'#message %mymac' substituted for %echo
                             -- and 'Hello' substituted for %mymac

#set mymac = 'World'         --'World' associated with mymac

#echo                        --'#message %mymac' substituted for %echo
                             -- and 'World' substituted for %mymac
```

If a single % had been specified in the first #set command, the macro %mymac would have been expanded (to the empty string) before defining the replacement text for the macro %echo. The double % results in the project system executing:

```
#message Hello
#message World
```

The single % results in the project system executing:

```
#message ""
#message ""
```

Setting Macro Values

#set

#set macroname = string

The #set command associates a macro name with a string. Any previous setting for the given macro is lost. The macro name in the #set command should *not* be delimited by % characters. The string *should* be enclosed in single quotes if it contains embedded spaces or project system keywords.

For example:

```
#set cwindow = TopSpeed
#set linkit = '#link myfile'
...
#if '%cwindow'= TopSpeed #then
    #pragma link(CS_GRAPH.LIB)
#endif

%linkit
```

#expand <file-name>

The filename is subjected to redirection analysis, and the following macros are defined:

%cpath	Is set to the fully expanded filename where the file would be created.
%opath	Is set to the fully expanded filename where the file would be opened.
%ext	Is set to the extension of the filename.
%tail	Is set to the filename, less extension, drive and path.
%cdir	Is set to the directory where the file would be created.
%odir	Is set to the directory where the file would be opened for read (if the file does not exist %opath is set the same as %cpath).

For example, suppose the redirection file has the line,

```
*.def : . ; c:\ts\include
```

and the file c:\ts\include\io.def exists, and the current directory is d:\test then,

```
#expand io.def
```

is equivalent to,

```
#set opath = d:\test\io.def
#set cpath = c:\ts\include\io.def
#set ext   = .def
#set tail  = io
#set odir  = d:\test\
#set cdir  = c:\ts\include\
```

#split <filename>

The filename is split into its base and extension. The following macros are defined:

<code>%ext</code>	Is set to the extension of the filename.
<code>%name1</code>	Is set to the filename, less extension.

For example:

```
#split d:\name.exe
```

is equivalent to,

```
#set ext   = exe
#set name  = d:\name
```

Special Project System Macros

A number of macros are used for special purposes by the Project System, and you should avoid defining macros of the same name inadvertently. Similarly, you should not define macros using trailing underbars.

The following is a list, in alphabetical order, of all such macros:

<code>%action</code>	Set to make, link, compile or run, depending on the mode of invocation.
<code>%cdir</code>	Set by the #expand command.
<code>%compile_src</code>	In compile mode, this is set to the name of the file to be compiled, with path and extension where available. Otherwise, it is set to the empty string.
<code>%cpath</code>	Set by the #expand command.
<code>%devsys</code>	Set by the Clarion environment to win.
<code>%editfile</code>	Set to the name of the file being edited in the topmost window. If no window is open, or in batch mode, it is set to the empty string.

<code>%editwin</code>	Set to the window number (0-9) of the topmost window. If no window is open, or in batch mode, it is set to the empty string.
<code>%errors</code>	Count of errors produced by preceding compile or #file adderrors command.
<code>%ext</code>	Set by the #split and #expand commands.
<code>%filetype</code>	Set by the #system command to its second argument, and examined by the #link command.
<code>%jpicall</code>	Set by the #model command to its second argument, and examined by the #link command.
<code>%L</code>	Set by the #model command to either ‘’ (standalone) L (local) or ! (own). The #link command uses this to derive the name of any required library files.
<code>%link</code>	Set to the current link list.
<code>%link_arg</code>	Set to its argument by the #link command.
<code>%main</code>	Set to the assumed name of the main source file. In make or link mode when not using UNNAMED.PR, this is derived from the project filename, with path and extension removed. Otherwise, it is the supplied source filename complete with path and extension if specified.
<code>%make</code>	Set to on or off by the #compile , #link and #dolink commands, to indicate whether the target file was up to date.
<code>%manual_export</code>	Set this macro on to indicate that the #link command should not construct a .LIB file when a DLL is linked. If this macro is not specified, a .LIB file is created automatically from the corresponding .EXP file if found (see <i>Module Definition File</i> below), or from the object files in the link list.
<code>%model</code>	Set by the #model command to its first argument, and examined by the #link command.
<code>%name</code>	Set by the #split command.
<code>%obj</code>	Set to the object filename in a #compile command.
<code>%odir</code>	Set by the #expand command.

<code>%opath</code>	Set by the #expand command.
<code>%pragmastring</code>	Will always expand to the current state of the #pragma settings - this is useful for debugging.
<code>%prjname</code>	Set to the assumed name of the project - this is usually derived from the project filename, but with the path and extension removed. Where UNNAMED.PR is being used, it is derived from main source filename without source and extension.
<code>%remake</code>	Used within declare_compiler macros to determine whether source/object dependencies require a remake.
<code>%remake_jpi</code>	Used within declare_compiler macros to determine whether source/object dependencies require a remake. <code>%remake_jpi</code> should be used for object files created by TopSpeed compilers, which contain additional information.
<code>%reply</code>	Set by the #prompt command.
<code>%S</code>	Set by the #system command to either 16 or 32 indicating the instruction set being used to build the project. The #link command uses this to derive the name of any required library files.
<code>%src</code>	Set to the source filename in a #compile command.
<code>%system</code>	Set by the #system command to its first argument, and examined by the #model and #link commands.
<code>%tail</code>	Set by the #expand command.
<code>%tsc</code>	Set to on if a C or C++ source file is compiled.
<code>%tscpp</code>	Set to on if a C++ source file is compiled.
<code>%tsm2</code>	Set to on if a Modula-2 source file is compiled.
<code>%tspas</code>	Set to on if a Pascal source file is compiled.

The above macros are examined by the **#link** command to determine which libraries to include, and then set to off.

<code>%warnings</code>	Count of warnings produced by preceding compile or #file adderrors command.
------------------------	--

Basic Compiling and Linking

Specifying Compile and Link Options

Compile and link options are specified in a project file by means of the #system, #model and #pragma commands.

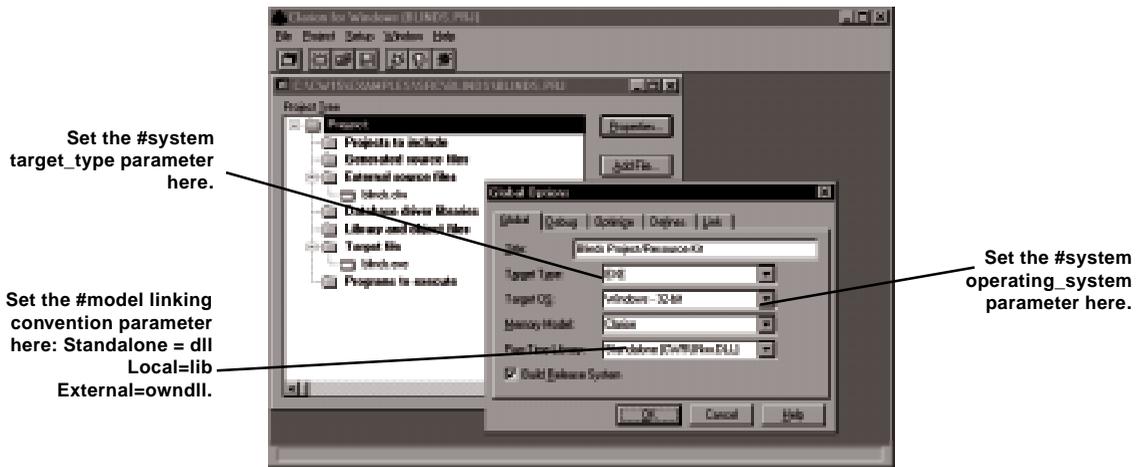
#system

#system operating_system [target_type]

The #system command is used to specify the target operating system and file type. The macros %system and %filetype are set to the first and second arguments. See *Special Project System Macros* below.

The first argument specifies the target operating system, and may be win or win32.

The second argument indicates the target file type, and may be exe, lib, or dll. If omitted, exe is assumed.



The #system command affects the behavior of subsequent #model and #link commands. Therefore a #system command must be specified before either of these. If more than one #system command occurs in a project, each must be followed by a #model command in order to take effect.

#model

#model memory_model [linking_convention]

The `#model` command is used to specify the memory model to be used for subsequent compiles and links. This memory model will continue to be used until modified by explicit `#pragmas`, or by another `#model` command.

The `#model` command sets the macros `%model` and `%jpicall` to its first and second parameters respectively. For example,

```
#model clarion dll
```

is equivalent to

```
#set %model = 'clarion'  
#set %jpicall = 'dll'
```

The first argument specifies the memory model, which is always `'clarion'` for Clarion projects. The second indicates the linking convention, which may be `dll`, `lib`, or `owndll`. If omitted, `dll` is assumed.

Setting the second argument to `dll` indicates that you will be creating an exe or dll that calls the standard Clarion dlls. Setting the second parameter to `lib` indicates that you will be creating an exe, lib or dll that includes all the components of the Standard Clarion libraries (and file drivers) in the exe, lib or dll. Using `owndll` indicates that you are linking to a dll that was previously created with the `lib` link convention, so the standard Clarion dlls are not linked.

The `#system` command must be specified before the first `#model` command.

#pragma

```
#pragma <#pragma> { , <#pragma> }
```

The `#pragma` command modifies the state of the `#pragma` options which affect the behavior of the TopSpeed compilers or linker. The syntax and meaning of all `#pragmas` are discussed under the *TopSpeed #pragmas* section below.

The special macro `%pragmstring` expands to the current state of all `#pragma` options which are not in their default state - this can be useful for determining exactly which options are being used for a given compile. For example:

```
#message '%pragmstring'
```

Compile and Link Commands

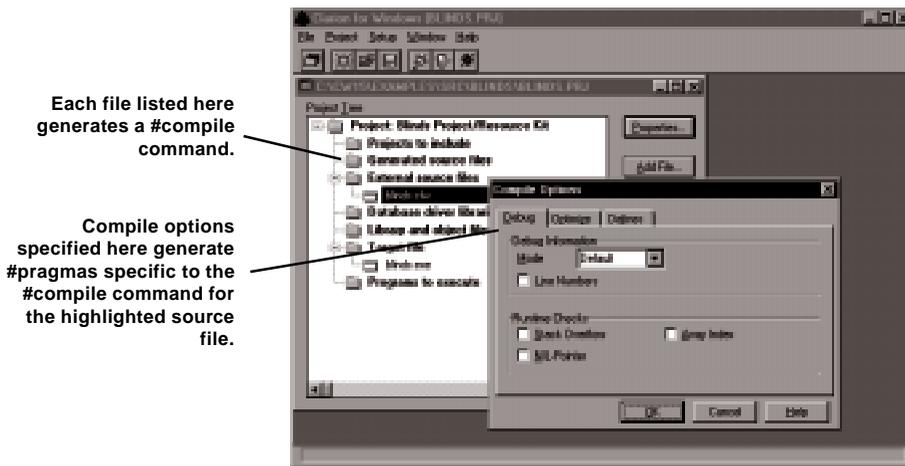
Whenever a file is compiled or linked, the current settings of the compiler or linker options (`#pragma` settings) are compared to those used when the file was last compiled or linked, to determine whether the file is up to date. If a compile or link is necessary, the current settings are passed on to the compiler or linker.

`#compile`

```
#compile<source> [ #to <object> ] [ / <#pragma> { , <#pragma> } ]
      { , <source> [ #to <object> ] [ / <#pragma> { , <#pragma> } ] }
```

The `#compile` command causes each nominated source file to be compiled (if necessary). The name of the object file may be specified using `#to`. If this is omitted, the name is derived from the source filename, with the extension `.obj`.

Any `#pragmas` specified in a `#compile` command apply only to the single source filename that precedes the `/` character.



The macro `%make` is set to on if a compile is necessary, off otherwise. The macros `%src` and `%obj` are set to the names of the source and object filenames.

Each object file is added to the link list, i.e. there is an implicit:

```
#pragma link( %obj )
```

For example:

```
#compile fred.c #to fred.obj
#compile george.cpp /debug(vid=>full)
```

It is possible to reconfigure the behavior of the Project System when compiling source files of a given extension using the `#declare_compiler` command. This may also be used to declare actions to perform for different

file extensions - for example, to support third-party compilers or preprocessors. See *Other Commands* below.

#link

#link <target_filename>

The `#link` command links together (if necessary) all the files in the link list to the nominated executable or library file. The file type is determined by the extension of the nominated target file, or, if there is no extension, by the file type specified in the most recent `#system` command. If neither are specified, the default is to produce an executable file. The effect of `#link` is to set the macro `%link_arg` to the specified filename.

The Project System maintains a list of those files which are to be used as input to the linker the next time an executable or library file is created. This list is known as the *link list*. A filename may be added to the link list using the `#pragma link` command.

For example:

```
#pragma link (mylib.lib)
```

However, it is seldom necessary to use `#pragma link` explicitly, as all the TopSpeed compilers add the resulting object file to the link list whenever a source file is compiled using `#compile`. In addition, when the `#link` command is encountered, all required standard library files, and other object files which are imported by those already on the link list are also added to the list. The link list is cleared after each link.

The `#link` command differs from the similar `#dolink` command in that (so far as the Project System can determine), any additional object files required are automatically added to the link list before linking. This includes any TopSpeed library files, and also (with an implicit `#autocompile` command) all modules imported with `IMPORT` clauses in TopSpeed Modula-2 or with `#pragma link` statements in TopSpeed C or TopSpeed C++ source files. In addition, `#link` will determine from the target file type any additional processing that needs to be applied to the output file.

For certain specialized requirements, the use of `#link` may be inappropriate—for example, if a specialized startup file is required, or when building library files, where explicit control of exactly which files are included may be preferred. In such cases, the `#dolink` command should be used.

#dolink <target_filename>

The `#dolink` command takes the object files which have previously been added to the link list, and combines them into an executable or library file (depending on the extension of the nominated target file), if required to keep the target file up to date. No additional files are added to the link list, so all required files must have been specified previously, by means of `#pragma`

link, #pragma linkfirst, #compile, and #autocompile. For simple projects, the use of #link is preferable because the link list is dynamically maintained by the project system, freeing the developer from this responsibility.

When finished, the #dolink command clears the link list.

See also:

`#pragma link_options (link)`

#autocompile

The #autocompile command examines the object files which are currently in the link list, to see which objects they need to be linked with. This would include objects specified using a #pragma link in a TopSpeed C or C++ source file, or in the case of module based languages such as TopSpeed Modula-2 imported modules.

Each resulting object file, which is not already in the link list, is then compiled (if necessary) and added to the link list. If there is more than one possible source for a given object file, an error is reported. This process is repeated until the link list stops changing.

It is not necessary to use #autocompile for simple projects where #link is used rather than #dolink, as #link performs an implicit #autocompile.

#ignore

`#ignore <filename>`
`#ignore #pragmastring`

There are two forms of the #ignore command. The first, where a filename is specified, tells the Project System to ignore the date of the nominated file when deciding whether or not to compile. This is useful when a 'safe' change is made to a widely used header file, to prevent mass recompile.

The special form #ignore #pragmastring directs the Project System to ignore the #pragma settings when deciding whether or not to compile a file. This may be useful, for example, when a new compile-time macro has been defined, but there is no need to recompile everything.

#implib

`#implib <libfilename>`

The #implib command is used to create (if necessary) a dynamic link library file. There are two forms of this command, which operate in slightly different ways. If a single filename is specified, this names an import library file, which is created (if not up-to-date) from the object files in the link list. The object files are scanned and each public procedure or variable is exported. For example:

```
#pragma link( fred.obj, joe.obj )  
#implib mylib.lib
```

In the second form of the `#implib` command, an import library filename and a module definition file (`.exp`—see *Module Definition File* below) are both specified, and the library file is created (if not up-to-date) from the symbols named in the definition file. This form of the command is equivalent to using the `tsimplib` utility that comes with TopSpeed C, C++, and Modula-2.

#implib <expfilename> <libfilename>

Using `#implib` in the second form requires you to create and maintain the list of exports 'by hand', whereas the first form exports all public names automatically. The use of a module definition file is an advantage if you need to maintain compatibility with previous versions of an interface, and it also allows you to export only the procedures which need to be exported.

When `#implib` is used with a module definition file, the link list is cleared.

Conditional Processing and Flow Control

Project file commands may be executed conditionally, using `#if`, `#then`, `#elseif`, `#else`, and `#endif` commands. In addition, processing may be stopped with `#error` and `#abort` when certain conditions occur.

#if

The syntax of the `#if` command is as follows :

```
#if <boolean-expression> #then
  commands
#elif <boolean-expression> #then
  commands
#else
  commands
#endif
```

The `#elseif` part may be omitted, or may be repeated any number of times. The `#else` part may be omitted.

The expressions are evaluated in order, until one of them yields true, then the following command sequence is executed. If none of the expressions yield true, and the `#else` part is present, then the commands following `#else` are executed. All other commands are ignored.

The syntax and semantics of boolean expressions are described under *Boolean Expressions* below.

#error

#error <string>

This command terminates the current project. Under the Clarion environment, the Text Editor is opened at the position of the `#error` command, and displays the supplied string as the error message. For example:

```
#if "%name"="" #then
  #error "name not set up"
#endif
```

#abort

#abort [on | off]

This command is used to control whether a failed `#compile` or `#run` command will terminate a project. If abort mode is on, a project will be aborted as soon as a `#compile` fails, or a `#run` command produces a non-zero return-code. If abort mode is off, a project will only be aborted if an internal command fails, including a `#link`, `#implib` or `#exemod` command.

`#abort on` will set abort mode to on, while `#abort off` will turn it off. `#abort` without one of the above arguments will abort the current project immediately.

The default abort mode is on when running under the Clarion environment.

User Interface

The following commands allow you to collect information and provide feedback during the make process.

#message

#message <string>

This command displays the specified string in the make display window. This can be used to indicate progress through the project file, or to display status messages. For example:

```
#message "finished making %prjname"
```

#prompt

#prompt <promptstring> [<defaultstring>]

This command prompts you to enter a string, by displaying the <promptstring> and waiting for a keyboard entry. The string you enter is returned as the value of the macro %reply. If <defaultstring> is specified, and no keyboard entry is made, the <defaultstring> will be used as the value returned to %reply. For example:

```
#prompt "Command line: " %cline  
#set cline = %reply
```

Boolean Expressions

Boolean expressions used in #if and #elsif commands are made up from the following boolean operators (listed in order of precedence):

```
#or  
#and  
#not  
=  
#exists  
#older  
( )
```

#or

```
boolean-expression = <factor> { #or <factor> }
```

A boolean expression containing one or more #or operators yields true if the evaluation of any of the factors yields true.

#and

```
<factor> = <term> { #and <term> }
```

A factor containing one or more #and operators yields false if the evaluation of any of the terms yields false.

#not

```
<term> = #not <term>
```

A term preceded by the #not operator yields true if the evaluation of the term yields false, and vice versa.

= (comparison)

```
<term> = string = string
```

A term containing a comparison operator yields true if the strings are identical, otherwise false. == may be used instead of =.

The = operator and second string may be omitted, in which case the first string is compared against the string “on”. That is,

```
DemoSwitch =
```

is equivalent to

```
DemoSwitch = “on”
```

The first string may be replaced by an expression of the form name1(name2), where name2 names a #pragma of class name1. In this case, the expression is replaced by the current setting of the specified #pragma, before the comparison is made.

#exists

```
<term> = #exists <file-name>
```

A term containing the #exists operator yields true if the file exists (after applying redirection to the filename), otherwise false.

#older

```
<term> = <file-name> #older <file-name> { , <file-name> }
```

A term containing the #older operator yields true if the first file specified is older than at least one of the other files specified, otherwise false.

Redirection is applied to all filenames (See *The Redirection File* below). This operator is often useful to determine whether a post/pre-processing action needs to be performed. For example:

```
#if mydll.lib #older mydll.exp #then
...
```

() Parenthesized boolean expressions

```
<term> = ( <boolean-expression> )
```

A term may consist of a *parenthesized boolean expression*, in order to alter or clarify the binding of other boolean operators. The term yields true if the enclosed boolean expression yields true. Arbitrarily complex boolean expressions may be formed.

The Redirection File

Clarion's development environment sets the working directory to the one in which the current .APP or .PRJ file resides. Additionally, Clarion uses the Redirection File to keep track of directories for the development environment's components. The name of the Redirection File is the same as the name of the executable which launches the Clarion development environment, with a .RED extension (*exename.RED*).

The Redirection File used for a specific application is determined by looking in the current directory (after changing directories if a .PR or .PRJ file has been loaded), then using the operating system PATH. Therefore, if there is a Redirection File in the same directory as the .APP or .PRJ file, it will be used for that application. Otherwise the first Redirection File encountered in the the operating system PATH is used.

The Redirection File tells the development environment where to find files and where to create new files. Each line of the redirection file is in the format:

```
filepattern = directory1 [;directory2]... [;directoryn]
```

The *filepattern* can be a file name or a file pattern using the standard DOS wild card characters: * and ?.

For example:

```
*.dbd = c:\clarion\obj
*.dll = .;c:\clarion\bin
*.lib = c:\clarion\obj;c:\clarion\lib
*.res = c:\clarion\obj;c:\clarion\lib
*.obj = c:\clarion\obj;c:\clarion\lib
*.rsc = c:\clarion\obj
*.* = .; c:\clarion\examples;c:\clarion\libsrc;c:\clarion\template
QCKSTART.TXA = c:\clarion\TEMPLATE
QCKSTART.TXD = c:\clarion\TEMPLATE
```

The first *directory* is the directory in which any *new* file of the type specified by the *filepattern* is created. This is only true for files created and saved by the development environment, such as .OBJ, .DBD, .LIB, .EXE, and .CLW. The subsequent directories are paths where Clarion will search for existing files. A period (.) specifies the working directory, that is, the one in which the current .APP or .PRJ file resides.

Note: Backup files are always created in the directory where the original file is located.

To edit the redirection file, choose **Setup ► Edit Redirection File**, or use any text editor. File patterns appear on the left, directory paths on the right.

Redirection Sections

The Redirection File can be separated into sections that are controlled by the project being made. The sections are of the form [NNNNN] where NNNNN is one of DEBUG16, DEBUG32, RELEASE16, RELEASE32, DEBUG, RELEASE, 16,32 and COMMON. Redirection lines that are within a section are only used if the section condition is true (COMMON is always true).

For example:

```
[DEBUG16]
*.obj = c:\test                #1
[DEBUG32]
*.obj = c:\test32              #2
[RELEASE]
*.obj = c:\release              #3
[COMMON]
*.* = work                       #4
```

In this example, if the project is 32 bit and the Build release flag is clear (not building a release system) then lines 1 and 3 are ignored and any .OBJ files are created in c:\test32 from line 2. If however the build release flag is then set, then any .OBJ files are created in c:\release from line 3 (for both 16 and 32 bit).

Redirection Macros

Redirection macros allow redirection files to be distributed between different machines. These can be set in the Clarion environment's .INI file and are expanded in the Redirection file wherever %name% is encountered within a target directory. The redirection values are listed in the Clarion .INI file section [Redirection Macros] in the form name=value. For example, with the INI file containing:

```
[Redirection Macros]
source = g:\cw\src
objbase = f:\cw
```

and the redirection file containing

```
*.* = %source%;%objbase%\obj;.
```

the actual redirection expands to:

```
*.* = g:\cw\src;g:\cw\obj;.
```

Root Macro

If not defined in the [Redirection Macros] section of the Clarion environment's .INI file, %ROOT% expands to the root Clarion directory—the directory containing the BIN program directory which contains the executable to launch the Clarion development environment. This allows the

Redirection File to be set up independent of the initial installation path using redirections of the form:

```
*.lib = %ROOT%\lib
```

File Management

These commands are all prefixed by the keyword `#file`, and handle the interface to the DOS/Windows file system with TopSpeed redirection built-in.

Filenames and Redirection Analysis

Filenames may be fully qualified, e.g. `C:\CW15\ORDERS\ORDER.TPS`, in which case, redirection analysis is not done. Alternatively, filenames may not be fully qualified, e.g. `ORDERS.TPS`, in which case redirection analysis is applied.

Redirection analysis means the project system compares the filename with the *filepatterns* in the current redirection file, until a match is found. Then, the project system searches only those directory paths associated with matching *filepattern* to locate the file. See *The Redirection File* above.

When creating new files, the project system creates the file in the first directory associated with the matching *filepattern*.

#file Commands

The following file system commands are available:

```
#file adderrors  
#file append  
#file copy  
#file delete  
#file move  
#file redirect  
#file touch
```

#file copy <src-filename> <dst-filename>

This command causes a file to be copied from `<src-filename>` to `<dst-filename>`. Both `<src-filename>` and `<dst-filename>` must be filenames without wildcard characters. Redirection is applied to both filenames.

#file delete <filename>

This command causes the nominated file to be deleted. `<filename>` must be a filename without wildcard characters. Redirection is applied to the filename.

#file move <src-filename> <dst-filename>

This command moves (renames) a file from <src-filename> to <dst-filename>. Both filenames must specify files on the same drive. Redirection is applied to both filenames.

#file touch <filename>

This command sets the date and time of <filename> to be the current date and time.

#file append <filename> <string>

This command appends the specified string to <filename>, followed by a CR/LF pair. The file will be created if it does not exist. This command can be used to build log files, etc.

#file redirect [<filename>]

This command changes the current redirection file to <filename>. If no filename is specified, then the command changes the current redirection file to the redirection file that began the project.

At the end of the project file, the redirection file is restored to the redirection file that began the project.

#file adderrors <filename>

This command processes the error messages in the nominated file, and adds them to the errors that will be reported when the project terminates.

Each error message must be in one of the following formats :

```
(filename lineno,colno): error string
(filename lineno): error string
filename(lineno): error string
```

To capture errors from a program with a different error format, a filter program can be used to translate them. For example:

```
#run 'masm %f; > %f.err'
#file adderrors %f.err
#run 'myprog %f; | myfilter > %f.err'
#file adderrors %f.err
```

If any errors are detected, and abort mode is on, the project will terminate and the errors will be reported in the make status window.

The macros %errors and %warnings are set to the number of errors and warnings detected.

Other Commands

#run <commandstring>

This command executes the command specified by <commandstring>. A #run command is generated whenever you add a file to the *Programs to execute* folder in the **Project Tree** dialog.

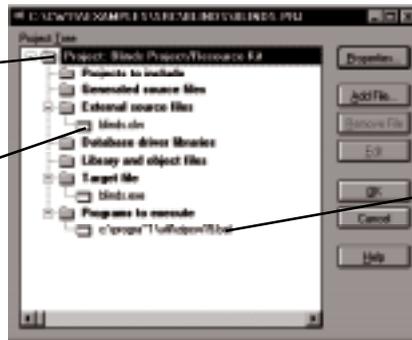
For example:

```
#run "dir > dir.log"
#run "myprog"
```

Note: Filenames within the command string (with the exception of the executable filename itself) are not automatically subject to redirection - #expand may be used before using #run if this is required.

Adding a file here generates a #include command.

Each file listed here generates a #pragma link command to add objects to the current link list.



Adding a file here generates a #run command.

#include <file-name>

A copy of the contents of the nominated file is inserted in the input stream. <filename> should specify a fully qualified filename, or an unqualified filename, in which case redirection analysis is applied (see *The Redirection File* above).

The current values of the link list, #pragma settings, and macros are fully available to the #include statements. In other words, the #include statements are handled as though they resided within the including .prj file.

#call <file-name>

A copy of the contents of the nominated file is inserted in the input stream. <filename> should specify a fully qualified filename, or an unqualified filename, in which case redirection analysis is applied (see *The Redirection File* above).

The current values of the link list, #pragma settings, and macros are *not* available to the #call statements, and the #call statements *cannot* modify these values in the calling environment. In other words, #call statements are handled as a process that is completely separate from the calling process.

#declare_compiler <file_extension> = <executed_macro>

This defines a macro which is invoked when compiling source files with an extension matching the first parameter. The macros %src and %obj, are set to the names of the source and object files.

Generally, you will not have to use this command explicitly, as all TopSpeed compilers are pre-declared in the Project System. For example the following is to invoke MASM

```
#declare_compiler asm=
'#set make=%remake
#if %%make #then
#edit save %%src
#expand %%src
#set _masmsrc=%opath
#expand %%obj
#set _masmobj=%cpath
#run "masm %%_masmsrc,%%_masmobj/MX/e; >masmtmp.$$$"
#file adderrors masmtmp.$$$
#file delete masmtmp.$$$
#endif
#pragma link(%%obj)'
```

#rundll <dll_name> <source_filename> <output_filename>

This command invokes an integrated TopSpeed compiler/utility. The first string is the DLL name, the second is the source filename, and the third is the output filename.

You should never have to use this command explicitly, as all TopSpeed compilers/utilities are pre-declared in the Project System.

#exemod

#exemod <file-name> <file-name> <file-name>

This command is the equivalent of using the tsexemod utility that comes with TopSpeed C, C++, and Modula-2. #exemod is required to make advanced overlay model programs, Windows programs and DOS DLLs. However, it is not necessary to use this command explicitly when making Windows programs.

TSEXEMOD is used to modify the header and segment information in a new format executable file (.EXE or .DLL), using the information in a module definition (.EXP) file. For example:

```
TSEXEMOD binfile.* expfile.exp mapfile.map
```

TopSpeed #pragmas

All TopSpeed languages, and the TopSpeed Project System, use a common set of compiler options known as *#pragmas*. In general, pragmas may appear in the source code or in a project file, and the effect will be the same. Pragmas may not appear in Clarion Language source code.

A pragma can be used in either the Project language, C++ code, or Modula-2 code. Some only work in certain places. A 'P' to the right of the pragma indicates it can be used in the Project language, a 'C' indicates it can be used in C++ code and an 'M' indicates it can be used in Modula-2 code.

Pragma Syntax

Modula-2 Pragma Syntax

Pragmas in TopSpeed Modula-2 occur in a special form of comment which begins with `(*#`. For example:

```
(*# check( index => off ) *)
```

Old-type Compiler Directives

In the original version of TopSpeed Modula-2, compiler directives starting with a \$ were used to specify compiler options. These directives are still accepted in later versions of TopSpeed Modula-2, with the following exceptions:

- ◆ \$B (Ctrl-Break handler). This is no longer supported. Use `Lib.EnableBreakCheck` instead.
- ◆ \$D (data segment name). This is supported, but adds the suffix `_BSS` (for uninitialized data) or `_DATA` (for initialized data) to the name instead of the `D_` prefix.
- ◆ \$J (use `IRET` instead of `RET`). This is not supported. Instead, you should use the pragma:

```
(*# call( interrupt => on ) *)
```

However, you may find that you have to make other changes as well as the effect of the pragma is different from the \$J directive:

- ◆ \$K (C calling convention). This is not supported. Instead, you should use the pragma:
- ```
(*# call(c_conv => on) *)
```
- ◆ \$M (code segment name). This is supported but adds the suffix `_TEXT` to the name instead of the `C_` prefix.
  - ◆ \$P (external names for local procedures). This is no longer supported. It is no longer applicable.

- ◆ \$Q (procedure tracing). This is no longer supported. Instead, you should use the pragma:
 

```
(*# debug(proc_trace => on) *)
```

 This enables a different method of tracing procedures. Refer to the `proc_trace` pragma for further details.
- ◆ \$X (80x87 stack spilling). This is no longer supported (and is no longer necessary).
- ◆ \$Z (NIL pointer checks). This still does NIL pointer checks but no longer clears memory.
- ◆ \$@ (preserve DS). This is no longer supported.

The support for these directives has been included with later systems so that your old programs and modules will recompile with minimum changes. However, you should avoid using the old directives with new programs, and use pragmas instead.

### **C and C++ Pragma Syntax**

Pragmas are an integral part of the C and C++ languages, and are implemented as compiler directives:

```
#pragma check(index => off)
```

### **Project System Pragma Syntax**

Pragmas in the TopSpeed Project System use a similar syntax to the C and C++ languages:

```
#pragma check(index => off)
```

Pragmas in the Project System may also be specified in the `#compile` command, to apply to a single compilation. For example:

```
#compile mandel.mod /debug(vid=>on)
```

## **Pragma Classes**

---

A `#pragma` takes the form `#pragma class(name=>value)`. The `#pragma` classes are as follows:

**Call #pragmas**

**Check #pragmas**

**Data #pragmas**

**Debug #pragmas**

**Define #pragmas**

**Expr #pragmas**

**Link and Linkfirst #pragmas**

**Link\_option #pragmas**  
**Module #pragmas**  
**Name #pragmas**  
**Optimize #pragmas**  
**Option #pragmas**  
**Project #pragmas**  
**Save and Restore #pragmas**  
**Warn #pragmas**

## Call #pragmas

---

#pragmas with the class name call affect all aspects of calling conventions, code segments, and code pointers. The current settings of the call #pragmas at the point at which a procedure's *definition* is encountered, determines the calling convention that is used to call the procedure. TopSpeed compilers detect if an inconsistent calling convention is used when a procedure is called. The type-safe linker reports an error if the calling conventions attributed to a given procedure do not match in every object file.

The following call #pragmas are available:

```
#pragma call(c_conv => on | off)
#pragma call(ds_entry => identifier)
#pragma call(ds_eq_ss => on | off)
#pragma call(inline => on | off)
#pragma call(inline_max => Number)
#pragma call(near_call => on | off)
#pragma call(o_a_copy => on | off)
#pragma call(o_a_size => on | off)
#pragma call(opt_var_arg => on | off)
#pragma call(reg_param => RegList)
#pragma call(reg_return => RegList)
#pragma call(reg_saved => RegList)
#pragma call(result_optional => on | off)
#pragma call(same_ds => on | off)
#pragma call(seg_name => identifier)
#pragma call(set_jump => on | off)
```

**#pragma call(standard\_conv => on | off)**

**#pragma call(var\_arg => on | off)**

**#pragma call(near\_call => on | off) cpm**

Specifies whether procedure calls are near or far. When on, the compiler calls procedures with near calls. The compiler can only use near calls if the calling and called procedures are in the same segment. The compiler checks that this is the case.

The default value is off. This example forces near calls:

```
#pragma call(near_call=>on)
```

**#pragma call(same\_ds => on | off) cpm**

Specifies whether to load the data segment (DS) register on entry to a procedure. When on, DS will not be loaded as part of the procedure prolog. This will only be correct when the DS setting of the calling procedure matches that of the called procedure. The compiler checks that this is the case.

This option is off by default. For example:

```
#pragma call(same_ds => on)
```

This stops DS from being loaded in the procedure prolog.

**#pragma call(c\_conv => on | off) cpm**

When on, this option enables the Microsoft C calling convention. In this convention, the compiler pushes procedure parameters in right to left order on the stack and the caller pops these parameters off the stack.

This is not the default, so you should only use this #pragma when interfacing to Microsoft C code. For example:

```
#pragma call(c_conv=>on)
```

You can also use the cdecl keyword in C and C++ to achieve the same effect.

See also the standard\_conv #pragma, which has the same effect for C and C++, but is ignored for Modula-2. The standard\_conv #pragma is set off by default.

**#pragma call(inline => on | off) cm**

If this #pragma is set on before a procedure definition, the compiler makes a copy of the procedure in the code rather than using a call instruction. The default value is off.

You can use this convention for any procedure, but this `#pragma` is mainly used together with the `reg_param` `#pragma` for simple machine-code procedures. For example:

```
#pragma save
#pragma call(inline => on, reg_param => (dx,ax))
static void outp(int port, unsigned char byt) =
{
 0xEE, /* out dx,al */
};
#pragma restore
```

makes `outp` an inline procedure, so a call to it appears as a single 80x86 machine instruction: `out dx,al`.

### **#pragma call(seg\_name => identifier) cpm**

Specifies the code segment name. `call(seg_name => nnn)` means that the compiler places the code for the procedure in segment `nnn_TEXT`. The default value depends on the memory model. In the small and compact models, the default is null; in the other models it is the name of the source file. For example, a code segment named `_TEXT` would be specified as:

```
#pragma call(seg_name => null)
```

and a code segment named `MYCODE_TEXT`, would be specified as:

```
#pragma call(seg_name => MYCODE)
```

The default setting is language dependant, and is not defined by the Project System.

### **#pragma call(ds\_entry => identifier) cpm**

This `#pragma` indicates a segment name which the DS register will point to throughout the execution of a procedure. If the identifier is null, the compiler names the segment `_DATA`. If the identifier is none, the compiler does not assume a fixed DS during procedure execution and uses DS as a general purpose segment register like ES.

```
#pragma call(ds_entry => MYDATA)
```

This example indicates that on entry to the procedure, DS will point to the segment named `MYDATA_DGROUP`.

### **#pragma call(reg\_param => RegList) cm**

TopSpeed languages pass procedure parameters in machine registers rather than using the stack. This generates smaller and faster code. This `#pragma` allows you to fine-tune individual procedure calls for maximum speed. Other vendors' languages use a less efficient calling convention; you must, therefore, disable this calling convention when interfacing to precompiled objects written for these compilers (see the *TopSpeed Advanced Programmer's Guide* that comes with TopSpeed C, C++, and Modula-2,

Chapter 5: Multi-language Programming). This `#pragma` has no effect on structure parameters, which are always passed on the stack.

The argument for `reg_param` is a register list, specifying which registers should be used. Registers for parameters are allocated left to right from the list. The table shows how the compiler allocates registers dependent on parameter types:

|                |                                                                                       |
|----------------|---------------------------------------------------------------------------------------|
| 1 byte         | ax, bx, cx, dx                                                                        |
| 2 bytes        | ax, bx, cx, dx, si, di                                                                |
| 4 bytes        | ax, bx, cx, dx, si, di for low word.<br>ax, bx, cx, dx, si, di, es, ds for high word. |
| floating point | st0, st1, st2, st3, st4, st5, st6                                                     |

Note that the `es` and `ds` registers will only be used for the high word of a 4-byte parameter where that parameter is of pointer type. If either the low or high word cannot be allocated, then the whole parameter is passed on the stack.

When the compiler exhausts the list of registers, it passes the parameter on the stack. If you specify an empty list, the compiler uses the stack for all parameters.

The default setting for the TopSpeed calling convention is:

```
#pragma call(reg_param=>(ax,bx,cx,dx,st0,st6,st5,st4,st3))
```

The default setting for the stack calling convention is:

```
#pragma call(reg_param => ())
```

### **`#pragma call(reg_saved => RegList)` cm**

This `#pragma` specifies which registers a procedure preserves. The argument `RegList` is a list that specifies the set of registers.

The default set for the TopSpeed calling convention is:

```
ccall(reg_saved=>(ax,bx,cx,dx,si,di,ds,st1,st2))
```

The default set for the stack calling convention is:

```
#pragma call(reg_saved=>(si,di,ds,st1,st2))
```

### **`#pragma call(o_a_size => on | off)` m**

When on, this option passes the size of open array parameters on the stack:

```
(*# call(o_a_size => on) *)
```

This `#pragma` has no effect for *value* parameter open arrays, unless the `o_a_copy` `#pragma` is set off.

The default setting is on.

**#pragma call(o\_a\_copy => on | off) m**

When on, open array parameters are copied onto the stack as part of the procedure prolog. If off, only a reference to the array is passed. Note that the open array parameters size must be passed in order for a copy to be made - see #pragma call(o\_a\_size). The default setting is on.

**#pragma call(ds\_eq\_ss => on | off) m**

It controls whether VAR parameters use 16- or 32-bit pointers. The default setting is on for small and medium models, otherwise off.

**#pragma call(var\_arg => on | off) m**

When on, it implies that the following procedures take a variable number of arguments. This effectively disables the “too many arguments” error that the compiler would normally detect. The consequence however, is that the compiler cannot carry out any type checking on the arguments.

This #pragma should be used when calling C procedures (such as printf) where the number of arguments varies:

```
(*# call(var_arg => on,
 reg_param=>(),
 c_conv=>on) *)
```

The default setting is off.

**#pragma call(reg\_return => RegList) cm**

This #pragma is used to specify the registers to be used for return values of integer, pointer and floating point types. For example:

```
#pragma call(reg_return => (bx,cx))
```

The default setting is:

```
#pragma call(reg_return=>(ax,dx,st0))
```

**#pragma call(result\_optional => on | off) m**

It can be used to call a procedure as a proper procedure without generating a compiler error. For example:

```
(*# save *)
(*# module(result_optional => on) *)
PROCEDURE FuncProc(x: CHAR): CARDINAL;
(*# restore *)
```

With this declaration, you can write both of the following:

```
i := FuncProc('a');
FuncProc('a');
```

This is only useful when the called procedure has a side effect that is more important than the result. It is particularly useful when calling TopSpeed C library procedures.

The default setting is off.

**`#pragma call(set_jump => on | off)` **cm****

This #pragma should only be used for the library routines which implement non-local jumps. The effect is to inform the compiler of the non-standard register saving properties of these routines.

**`#pragma call(inline_max => Number)` **cpm****

This #pragma controls the largest procedure which is inlined. The default setting is 12, which corresponds to the minimum code size for most programs. A larger value increases the code size and may accelerate code execution.

The #pragma takes effect for each call, so a procedure may be called in different ways at different places.

**Note:** Procedures are not inlined if the body has not been compiled before the call.

**`#pragma call(standard_conv => on | off)` **c****

The effect on C and C++ programs is the same as the call(c\_conv) #pragma. For Modula-2 there is no effect. The default is off.

**`#pragma call(opt_var_arg => on | off)` **cp****

This #pragma controls whether optimized entry sequences are generated for procedures with variable parameter lists. The default is on.

## Data #pragmas

---

#pragmas with the class name data affect data segmentation, data pointers and all aspects of data layout. The current settings of the data #pragmas at the point of a variable's declaration will affect the way in which it is accessed.

The following data #pragmas are available:

**`#pragma data(c_far_ext => on | off)`**

**`#pragma data(class_hierarchy => on | off)`**

```

#pragma data(compatible_class => on | off)
#pragma data(const_assign => on | off)
#pragma data(const_in_code => on | off)
#pragma data(cpp_compatible_class => on | off)
#pragma data(ext_record => on | off)
#pragma data(far_ext => on | off)
#pragma data(near_ptr => on | off)
#pragma data(packed => on | off)
#pragma data(seg_name => identifier)
#pragma data(stack_size => Number)
#pragma data(threshold => Number)
#pragma data(var_enum_size => on | off)
#pragma data(volatile => on | off)
#pragma data(volatile_variant => on | off)

```

### **#pragma data(seg\_name => identifier) cpm**

The `#pragma data(seg_name=>xxx)` specifies that the compiler should place global initialized data objects in a segment named `xxx_DATA`, and global uninitialized data objects in a segment named `xxx_BSS`. These both have group name `xxx` and are in the `FAR_DATA` class. If the size of a data object is larger than the global data threshold, the compiler places the object in a separate segment.

The following example makes the names of the default segments: `MYDATA_DATA` and `MYDATA_BSS`. These segments are in group `MYDATA` and have class `FAR_DATA`:

```
#pragma data(seg_name => MYDATA)
```

You can also specify `null`, to indicate the names `_BSS` and `_DATA`. The default value is `null` in all models except for extra large and multi-thread. For example:

```
#pragma data(seg_name => null)
```

### **#pragma data(far\_ext => on | off ) cp**

When `on`, the code generator does not assume that external variables are in the segment specified by the segment `#pragma`. The `#pragma` defaults to `on`. For example:

```
#pragma(seg_name=>MYDATA, far_ext=>off)
```

makes the name of the default segments MYDATA\_DATA and MYDATA\_BSS in group MYDATA. The compiler assumes external data objects to be in the same segment.

### **#pragma data(c\_far\_ext => on | off) pm**

When on, the code generator does not assume that external variables are in the segment specified by the seg\_name #pragma. The #pragma defaults to off in all memory models. For example:

```
(*# data(seg_name => MYDATA, c_far_ext => off) *)
```

makes the name of the default segments MYDATA\_DATA and MYDATA\_BSS in group MYDATA. This #pragma is not particularly useful in Modula-2 except for interfacing to C.

### **#pragma data(near\_ptr => on | off) cm**

Specifies whether data pointers are near or far. This #pragma also affects pointers generated by the & operator and by implicit array to pointer conversions in C and C++. For example:

```
#pragma data(near_ptr => on)
```

makes data pointers near.

### **#pragma data(volatile => on | off) m**

Variables declared when this #pragma is set to on are considered to be volatile, and will always be kept in memory, rather than being kept in registers across statements.

The default setting is off. This #pragma is not allowed in a project file, and is not available for C and C++, where the volatile keyword should be used.

### **#pragma data(volatile\_variant => on | off) m**

The effect is as for #pragma data(volatile), but applies to variables of variant record types only. The default setting is off.

### **#pragma data(ext\_record => on | off) pm**

Normally TopSpeed does not allow two fields in different alternatives of a variant record to have the same name. Using this #pragma:

```
(*# data(ext_record => on) *)
```

will allow you to use the same name in different alternatives, if the fields are located at the same offset in the variant record and they have the same data type.

The default setting is off.

**#pragma data(var\_enum\_size => on | off) pm**

Enumeration constants with less than 256 alternative values are normally stored in one byte. Switching this option off:

```
(*# data(var_enum_size => off) *)
```

will force the compiler to store them as two-byte quantities. This is particularly useful for interfacing to third-party libraries and operating system calls that expect a word value. Without this #pragma the enumeration would be byte rather than word size.

The default setting is on.

**#pragma data(stack\_size => Number) cm**

Specifies the size of the stack. You must place this #pragma in the file containing the main procedure (the main module in Modula-2). If the stack size cannot be set to the specified size, the compiler uses the largest possible size. The default size is 16K bytes. For example:

```
#pragma data(stack_size => 0x6000)
main()
{
 /* statements */
}
```

makes the size of the run-time stack 0x6000 bytes (24K).

**#pragma data(packed => on | off ) pm**

This #pragma controls whether record fields are packed at bit level. The default setting is off.

**#pragma data(const\_in\_code => on | off) p**

This #pragma controls whether constants are put in to a code or data segment. The default setting is on.

**#pragma data(class\_hierarchy => on | off ) pm**

This #pragma controls whether information about class hierarchies is included in the class descriptor (method table). The information is used by the IS operator and TypeGuards with check on. The default setting is on.

**#pragma data(cpp\_compatible\_class => on | off ) pm**

This #pragma controls whether the compiler includes extra information in class descriptors to provide compatibility with C. The default setting is off.

**#pragma data(compatible\_class => on | off)                      cp**

This #pragma controls whether the compiler includes the correct information in class descriptors to provide compatibility with Modula-2. The default setting is off.

**#pragma data(threshold => Number)                                      cpm**

This #pragma sets the global data threshold. This determines at what size a data object is placed in a segment of its own. The default setting is 10000 bytes.

**#pragma data(const\_assign => on | off )                              pm**

This #pragma controls whether it is possible to assign to a structured constant. The default setting is off.

**Note:** If `const_in_code=>on` is specified, assignments to constants will result in protection violations.

## Check #pragmas

---

#pragmas with the class name check control run-time error checking. These can help you to locate erroneous program logic, but at the expense of slower execution. All these #pragmas default to off.

When a run-time check detects an error, the default action is to terminate the process and create the file CWLOG.TXT.

The following check #pragmas are available:

**#pragma check(guard => on | off)**

**#pragma check(index => on | off)**

**#pragma check(nil\_ptr => on | off)**

**#pragma check(overflow => on | off)**

**#pragma check(range => on | off)**

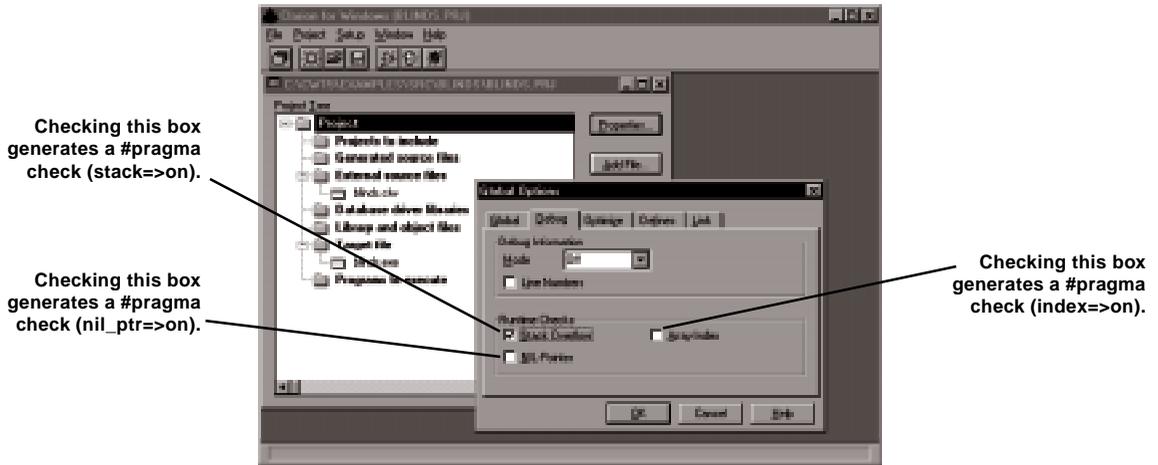
**#pragma check(stack => on | off)**

**#pragma check(stack => on | off) cpm**

When on, the run-time system checks that your program does not run out of stack space. You can increase the size of the stack using the data(stack\_size) #pragma.

**#pragma check(nil\_ptr => on | off) cpm**

When on, the run-time system checks for any dereference of NULL or NIL pointers.

**#pragma check(range => on | off) pm**

When on, a range check is performed whenever a value is assigned to a variable of subrange or enumerated type. In addition, compile-time values are checked to ensure that they are in the range of their type.

**#pragma check(overflow => on | off) pm**

When on, the run-time system checks that numeric values do not go out of range.

**#pragma check(index => on | off) cpm**

When on, the run-time system checks for the use of an array index larger than the array size.

**#pragma check(guard => on | off) pm**

This #pragma controls whether checks are performed on the checked-guard operator.

## Name #pragmas

---

#pragmas with the class name control aspects of linkage naming. However, the C programmer should also be familiar with C name mangling and extern declarations.

The following name #pragmas are available:

**#pragma name(prefix => (none | modula | c | os2\_lib | windows))**

**#pragma name(prefix => string)**

**#pragma name(upper\_case => on | off)**

**#pragma name(upper\_case => on | off) cp**

This #pragma is available in C and C++ Only. It specifies whether public names should be converted to upper case. You would use this when interfacing to Pascal, or to third party C libraries. The default setting is off.

**#pragma name(prefix) cpm**

There are two forms of this #pragma:

In Modula-2      **name(prefix => (none | modula | c | os2\_lib | windows))**

In C and C++      **name(prefix => string)**

**#pragma name(prefix => (none | modula | c | os2\_lib | windows)) mp**

This #pragma is available under Modula-2 (under C and C++ the syntax is slightly different - see #pragma name(prefix => string).

The name(prefix) #pragma specifies the prefix and case of the public names that the compiler uses. The public names are names for non-static procedures and external data objects. By default, TopSpeed Modula-2 prefixes all external names with the name of the module followed by an '@' for data and a '\$' for procedures. You will need to use this #pragma to interface to TopSpeed C.

The prefix #pragma specifies which prefix scheme to use:

modula            Use the TopSpeed Modula-2 naming convention of prefixing all external names with the name of the module and an '@' or a '\$'.

none              Puts no prefix on external names.

|         |                                                                                 |
|---------|---------------------------------------------------------------------------------|
| c       | Use the C naming convention (adds an underbar, ‘_’ to all external names).      |
| os2_lib | Use the OS/2 library standard (prefix all external names with the module name). |
| windows | Use the Microsoft Windows external naming convention.                           |

### **#pragma name(prefix => string) cp**

This #pragma is available under C and C++ Only. Under Modula-2 the syntax is slightly different - see #pragma name(prefix => (none | modula | c | os2\_lib | windows)).

The value is a string specifying the prefix to all public names. An empty string specifies no prefix. The default prefix is an underbar.

If you wish to interface to TopSpeed Modula-2, you can use this #pragma to specify the module prefix with a dollar (\$) suffix. For example, to use the WrStr procedure from module IO:

```
#pragma name(prefix => "IO$")
void WrCard(unsigned);
```

In C, a Pascal or Modula2 linkage specification can specify a module name within the linkage specification, in which case the use of this #pragma is not necessary.

The default setting is language-dependent. The Project System does not set a default value for this macro.

## Optimize #pragmas

---

#pragmas with the class name optimize control optimizations performed by the TopSpeed code generator. By default, all optimizations are enabled. Turning off an optimization will result in poorer code quality, and is unlikely to have a significant impact on compile times.

The following optimize #pragmas are available:

```
#pragma optimize(alias => on | off)
#pragma optimize(const => on | off)
#pragma optimize(cpu => 86 | 286 | 386 | 486)
#pragma optimize(cse => on | off)
#pragma optimize(jump => on | off)
```

**#pragma optimize(loop => on | off)**

**#pragma optimize(peep\_hole => on | off)**

**#pragma optimize(regass => on | off)**

**#pragma optimize(speed => on | off)**

**#pragma optimize(stk\_frame => on | off)**

**#pragma optimize(cse => on | off)** **cpm**

When on, the compiler minimizes evaluation of complete expressions by keeping partial results in a temporary register. The default setting is on.

**#pragma optimize(const => on | off)** **cpm**

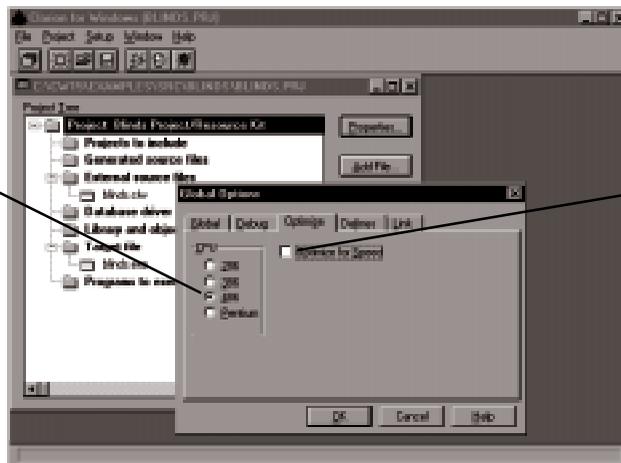
When on, the compiler will hold frequently used constants in registers to produce faster code. The default setting is on.

**#pragma optimize(speed => on | off)** **cpm**

When on, TopSpeed tries to make the code run as fast as possible without regard for the code size. When off, TopSpeed tries to make the code as small as possible.

A good example of the difference between optimizing for speed and optimizing for space is the use of a for loop. When optimizing for speed, the compiler might use nop instructions to place jump labels inside the for loop on even boundaries. The 80x86 architecture makes this much quicker than odd boundaries, but each nop adds another byte to the code size. This means that when optimizing for space, the compiler eliminates the extra nop instructions. The default setting is on.

Checking this box generates a #pragma optimize (cpu=>486).



Checking this box generates a #pragma optimize (speed=>on).

**#pragma optimize(stk\_frame => on | off) cpm**

When on, the compiler will only make stack frames where required, thus eliminating the need to set up the BP register. This optimization can only be made when all parameters and local variables for a procedure can be held in machine registers.

When off, the compiler always sets up the BP register, thus allowing a complete activation stack listing while debugging. The default setting is on.

**#pragma optimize(regass => on | off) cpm**

When on, the compiler spends time finding the best allocation of registers for variables. This results in fast and tight code but slower compile. The default setting is on.

**#pragma optimize(peep\_hole => on | off) cpm**

When on, the compiler performs a variety of machine-code translations, generating smaller and faster code. The default setting is on.

**#pragma optimize(jump => on | off) cpm**

When on, the compiler will rearrange loops to eliminate as many jumps as possible, thus generating faster code. The default setting is on.

**#pragma optimize(loop => on | off) cpm**

When on, the compiler uses the loop depth when eliminating common sub-expressions and performing jump optimizations. The result of this optimization is faster, but potentially larger, code. The default is on.

**#pragma optimize(alias => on | off) cpm**

When on, this allows the compiler to assume that variables in a procedure will not also be used indirectly with a pointer in the same procedure. This assumption is not strictly allowed in ANSI C but is correct for all meaningful programs. The default setting is on.

**#pragma optimize(cpu => 86 | 286 | 386 | 486) cpm**

This controls the instructions used by the code generator, by declaring the processor to be used. The default is `cpu=>286`. This is normally set on the Project System's Optimize tab.

## Debug #pragmas

#pragmas with the class name *debug* control the amount of additional information produced by the code-generator to assist debugging programs.

The following debug #pragmas are available.

**#pragma debug(line\_num => on | off)**

**#pragma debug(proc\_trace => on | off)**

**#pragma debug(public => on | off)**

**#pragma debug(vid => off | min | full)**

**#pragma debug(vid => off | min | full) cpm**

When full, the compiler places information for the TopSpeed Visual Interactive Debugger (VID) into a .DBD file. Use this option when debugging your program with the TopSpeed debugger.

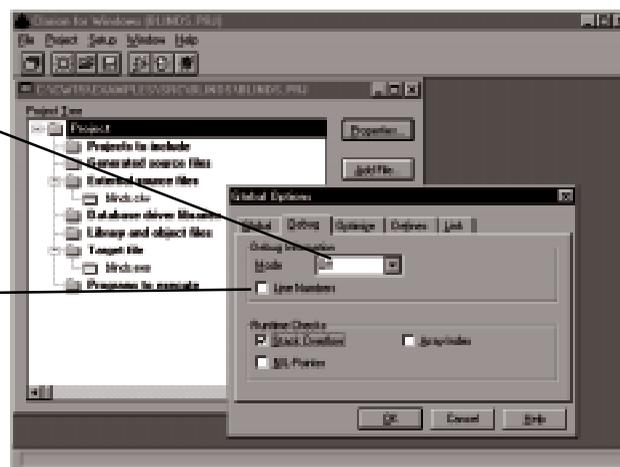
**Note:** This #pragma disables the register usage and stack frame optimizations, allowing full access to variables within the debugger. All local variables are treated as volatile, to ensure that their values are not held in registers across statements, thus ensuring that the debugger can access their values at all times.

When min, the compiler performs the optimizations described above, and does not treat local variables as volatile. The debugger can still be used, but cannot reference local variables and some stack frames.

When off, the compiler generates no debugger information, thus speeding compile, generating the best possible code, and saving disk space. The default setting is off.

This drop down list controls the #pragma debug(vid=>).

Checking this box generates a #pragma debug(line\_num=>on).



**#pragma debug(proc\_trace => on | off) pm**

When this #pragma is on, the compiler generates instructions to call the procedures EnterProc and ExitProc on, respectively, entering and exiting every procedure. These procedures can then perform any procedure tracing you may require.

**Note:** You should ensure that this #pragma is off for the EnterProc and ExitProc procedures themselves, otherwise infinite recursion will occur and your program will undoubtedly crash.

The two procedures must be *visible* to the module in which proc\_trace is set on. This means that the module itself must define the procedures ExitProc and EnterProc or the module must specifically import them using an unqualified import.

The default setting is off.

**#pragma debug(line\_num => on | off) cpm**

This #pragma causes the compiler to generate line number information for debuggers such as symdeb. This information is stored in object files and printed in the map file. The default setting is off.

**#pragma debug(public => on | off) cpm**

This causes private objects to be made public to facilitate the use of debuggers such as symdeb. It may cause duplicated public warnings at link-time in languages such as C and C++ which do not have a modular structure. These warnings may be safely ignored, although it is recommended that such procedures should be renamed to avoid possible confusion. The default setting is off.

## Module #pragmas

---

#pragmas with the class name module control options that apply to an entire source file or module. These #pragmas should be specified at the top of any source files to which they apply, or in the project file.

The following module #pragmas are available:

**#pragma module(implementation => on | off )**

**#pragma module(init\_code => on | off )**

**#pragma module(init\_prio => Number)**

**#pragma module(smart\_link => on | off)**

**#pragma module(init\_code => on | off ) pm**

When on, it implies that the module contains initialization code to be run when the program is loaded and before the main module is executed. Switching the option off is useful for modules written in other languages, as it will stop the linker warning of undefined symbols:

```
(*# module(init_code => off) *)
```

**Note:** If an implementation module sets this #pragma off, then there is a knock-on effect, i.e., all imported modules must also have init\_code set to off.

The default setting is on.

**#pragma module(implementation => on | off ) pm**

This #pragma specifies whether or not a definition file (.DEF or .ITF) has a corresponding object file. It should be turned off if the definition file defines interfaces to routines in a different language, to prevent the Project System from attempting to remake the corresponding object file. The default is on.

This #pragma can also be used in the implementation part of a module, before any module source code. In this case it overrides the default naming of the associated object file. Normally the name of the .OBJ file corresponding to a module is taken from the module name. When this #pragma is set off, the object filename will be taken from the filename, not the module name.

**#pragma module(smart\_link => on | off) cpm**

Setting this #pragma to off disables the smart linking feature, to the extent that either all or none of the objects in each segment from a compile will be included in a link. This may result in quicker linking, and also may allow other linkers (such as Microsoft) to be used. (There are many potential problems with trying to use a non-TopSpeed linker, and it is definitely not recommended). The default setting is on.

**#pragma module(init\_prio => Number) cp**

This #pragma is available under C only. It defines a priority for the initialization code for static objects. Normally the initialization order is undefined between files, but this #pragma allows you to control the initialization order in that files with higher priority are initialized before modules with lower priority. The number must be a value between 0 and 32. The default value is 16, and the C library uses values between 25 and 32 (it is therefore not recommended to use values in this range, otherwise part of the library may not have initialized before user code is executed).

## Option #pragmas

---

#pragmas with the class name option control language-dependent options, such as TopSpeed extensions. The following option #pragmas are available:

```
#pragma option(ansi => on | off)
#pragma option(bit_opr => on | off)
#pragma option(incl_cmt => on | off)
#pragma option(lang_ext => on | off)
#pragma option(min_line => on | off)
#pragma option(nest_cmt => on | off)
#pragma option(pre_proc => on | off)
#pragma option(uns_char => on | off)
```

### **#pragma option(ansi => on | off) cp**

This #pragma is available under C and C++ Only. If it is set to on, ANSI keywords only are allowed. The default setting is off.

### **#pragma option(lang\_ext => on | off) cp**

This #pragma is available under C and C++ Only. The following constructs are not valid under ANSI C, but are included in TopSpeed C and C++ when this #pragma is not set on:

- A type cast yields an lvalue if the operand is an lvalue.
- Procedures can be initialized with binary machine code.
- The relational operators (>, >=, <=, <) allow the operators to be a mixture of integer and pointer operands.
- Bitfields in C can have type char and unsigned char.
- Relative pointers.

The default setting is on.

### **#pragma option(nest\_cmt => on | off) cp**

This #pragma is available under C and C++ Only. When on, you can nest comments without causing an error message. For example:

```
/* This is a test comment
/* This is a nested comment */
*/
```

When off, nested comments cause an error message. The default is off, allowing the compiler to trap unterminated comments more easily and make it conform with ANSI C.

**#pragma option(uns\_char => on | off) cp**

This #pragma is available under C and C++ Only. When on, types declared as char lie between 0 and 255. When off, values declared as char lie between -127 and 128. The default setting is off.

**#pragma option(pre\_proc => on | off) cp**

This #pragma is available under C and C++ Only. When on, the compiler produces preprocessor output in a file with the same name but with extension .i. This output file makes it easy to debug the result of macro expansions. The default setting is off.

**#pragma option(incl\_cmt => on | off) cp**

This #pragma is available under C and C++ Only. When on, comments are preserved in preprocessor output. The default setting is off.

This #pragma has no effect unless #pragma pre\_proc is on.

**#pragma option(min\_line => on | off) cp**

This #pragma is available under C and C++ Only. When on, the preprocessor minimizes the number of blank lines in output. The default setting is on.

This #pragma has no effect unless #pragma pre\_proc is on.

**#pragma option(bit\_opr => on | off) pm**

This #pragma is available under Modula-2 only. It allows bitwise operations on cardinals:

( a AND/OR b, NOT a ).

The default setting is off.

## Warn #pragmas

---

#pragmas with the class name warn control the generation of compiler warnings. These #pragmas are only available under C and C++.

The warnings given by TopSpeed C and C++ help you to check, as far as possible, common coding mistakes. Since no compiler can determine your intentions, you may get warnings even if your code is correct. Your code may

generate some warnings more than others, so TopSpeed allows you to customize which warning checks are performed.

You can set each of the warning options to on, off, or err. When on, TopSpeed checks the code for that warning and reports the problem, but the problem does not stop the compile or linking. When off, TopSpeed ignores the warning. When err, TopSpeed checks the code for the warning, reports the problem, and does not allow linking until you have fixed the problem.

**Note:** TopSpeed C and C++ check the code and produce a warning for a good reason. Indeed, to use your non-ANSI C code, TopSpeed C uses a minimal set of warning messages by default. You should, therefore, think twice before turning off any of the default warning messages. We advise that you keep all the warnings either on or err.

The following warn #pragmas are available:

```
#pragma warn(wacc => on | off | err)
#pragma warn(wait => on | off | err)
#pragma warn(wall => on | off | err)
#pragma warn(watr => on | off | err)
#pragma warn(wcic => on | off | err)
#pragma warn(wcld => on | off | err)
#pragma warn(wcld => on | off | err)
#pragma warn(wcne => on | off | err)
#pragma warn(wcor => on | off | err)
#pragma warn(wcrt => on | off | err)
#pragma warn(wdel => on | off | err)
#pragma warn(wdne => on | off | err)
#pragma warn(wdnu => on | off | err)
#pragma warn(wetb => on | off | err)
#pragma warn(wfnd => on | off | err)
#pragma warn(wftn => on | off | err)
#pragma warn(wnid => on | off | err)
#pragma warn(wnre => on | off | err)
#pragma warn(wnrv => on | off | err)
#pragma warn(wntf => on | off | err)
#pragma warn(wovl => on | off | err)
#pragma warn(wovr => on | off | err)
#pragma warn(wpev => on | off | err)
```

```

#pragma warn(wpic => on | off | err)
#pragma warn(wpin => on | off | err)
#pragma warn(wpnd => on | off | err)
#pragma warn(wpnu => on | off | err)
#pragma warn(wprg => on | off | err)
#pragma warn(wral => on | off | err)
#pragma warn(wrfp => on | off | err)
#pragma warn(wsto => on | off | err)
#pragma warn(wtxt => on | off | err)
#pragma warn(wubd => on | off | err)
#pragma warn(wvnu => on | off | err)

```

**#pragma warn(wall => on | off | err) cp**

This #pragma affects the settings of all the warnings. If set to on or err, all warnings will be enabled.

**#pragma warn(wpcv => on | off | err) cp**

**Pointer conversion.** When on or err, the compiler checks for a conversion between two incompatible pointer types, or between a pointer and an integral type. The default setting is on.

**#pragma warn(wdne => on | off | err) cp**

**Declaration has no effect.** When on or err, the compiler checks for a declaration that has no meaning, for example, long int;. A declaration should contain a variable declarator, a structure or union tag, or members of an enumeration. The default setting is on.

**#pragma warn(wsto => on | off | err) cp**

**Storage class redeclared.** When on or err, the compiler checks that you have not declared the same variable differently within your program. For example:

```

int x; /* External linkage */
static int x; /* Internal linkage */

```

The static storage class always takes preference. The default setting is on.

**#pragma warn(wtxt => on | off | err) cp**

**Unexpected text in preprocessor command.** When on or err, the compiler checks for a new line character terminating a preprocessor command. The default setting is on.

**#pragma warn(wprg => on | off | err)** **cp**

**Unknown #pragma.** When on or err, the compiler checks for foreign #pragmas or mistakes in TopSpeed C #pragmas. If you are only creating code using TopSpeed C or C++ #pragmas, you should switch this warning to either on or err. The default setting is on.

**#pragma warn(wfnd => on | off | err)** **cp**

**Function not declared.** When on or err, the compiler checks for functions that have been called but not declared. If these functions occur, TopSpeed C assumes that the function is an extern function returning an int. The default setting is off.

**#pragma warn(wpnd => on | off | err)** **cp**

**Function prototype not declared.** When on or err, the compiler checks whether a function has a prototype associated with it. Prototypes are important to TopSpeed, since it can not do much type checking without them. You should, therefore, declare prototypes for all functions. It is best to keep this warning on or err. The default setting is off.

**#pragma warn(wnre => on | off | err)** **cp**

**No expression in return statement.** When on or err, the compiler checks for a return value in a non-void function. You should keep this warning off if you are compiling some old style C code without prototypes. The default setting is off.

**#pragma warn(wnrv => on | off | err)** **cp**

**No return value in function.** When on or err, the compiler checks for a return statement in a non-void function. The default setting is off.

**#pragma warn(watr => on | off | err)** **cp**

**Different const attributes.** When on or err, the compiler checks whether a function that expects a pointer to a variable gets a pointer to a constant. The default setting is on.

**#pragma warn(wftn => on | off | err)** **cp**

**Far to near pointer conversion.** When on or err, the compiler checks for conversion of a 32-bit far pointer to a 16-bit near pointer. The default setting is on.

**#pragma warn(wntf => on | off | err) cp**

**Near to far pointer conversion.** When on or err, the compiler checks for conversion of a 16-bit near pointer to a 32-bit far pointer. The default setting is on.

**#pragma warn(wubd => on | off | err) cp**

**Possible use of variable before assignment.** When on or err, the compiler checks that you have used a local variable before you have given it a value. TopSpeed checks this warning with a simple scan through the function, which can cause gotos and the like to generate false warnings.

**#pragma warn(wpnu => on | off | err) cp**

**Parameter never used in function.** When on or err, the compiler checks for a parameter that the code never uses, so declaration of dummy parameters generates warnings. The default setting is off.

**#pragma warn(wdnu => on | off | err) cp**

**Variable declared but never used.** When on or err, the compiler checks whether a local variable has been declared but never used in the function. The default setting is on.

**#pragma warn(wcne => on | off | err) cp**

**Code has no effect.** When on or err, the compiler checks statements and the left operand in a comma expression to see if they have no effect. The default setting is on. For example:

```
x y; /* expression has no effect */
f, x; /* left operand has no effect */
```

**#pragma warn(wclld => on | off | err) cp**

**Conversion may lose significant digits.** When on or err, the compiler checks for a conversion from long or unsigned long to int or unsigned int. The default setting is on.

**#pragma warn(wait => on | off | err) cp**

**Assignment in test expression.** When on or err, the compiler checks for a possible mistyping of the C equality (==) operator. The equality operator contains two =. For example:

```
if (x=y) printf("X equals Y"); /* is a mistake */
```

The default setting is on.

**#pragma warn(wetb => on | off | err) cp**

**Value of escape sequence is too large.** When on or err, the compiler checks that an escape sequence is in the range 0 to 255. The default setting is on.

**#pragma warn(wcor => on | off | err) cp**

**Value of constant is out of range.** When on or err, the compiler checks whether an integer constant is in the range of an unsigned long, or a floating point constant is in the range of a long double. The default setting is on.

**#pragma warn(wclt => on | off | err) cp**

**Constant is long.** When on or err, the compiler checks for an integral constant that has type long because of its value but does not have an L suffix. The default setting is off.

**#pragma warn(wral => on | off | err) cp**

**Returns address of local variable.** When on or err, checks for a return statement that returns the address of a local variable. This causes a problem because C reclaims the variable storage on completion of the function. The pointer, therefore, points at undefined data. The default setting is on.

**#pragma warn(wpin => on | off | err) cp**

**Default type promotion on parameter.** When on or err, the compiler compares the declaration of a parameter in an old-style function definition with the prototype for incompatible argument promotions. For example:

```
int func(char); /* parameter declared as char */

int func(IntegerByPromotion);
char IntegerByPromotion;
 /* INCOMPATIBLE */
{
 ...
}
```

**Note:** This is a violation of the ANSI C standard regarding compatible function declarations. The default setting is on.

**#pragma warn(wpic => on | off | err) cp**

**Parameter list inconsistent with previous call.** This warning is issued if a parameter declaration is incompatible with the corresponding parameter in a previous function declaration. The default setting is on.

---

**#pragma warn(wnid => on | off | err)** **cp**

**Address for local variable not in DGROUP.** When on or err, the compiler checks that a local variable does not have its address taken in small model, when using #pragma data(ss\_in\_dgroup => off). The default setting is on.

---

**#pragma warn(wrfp => on | off | err)** **cp**

**Function redeclared with fixed parameters.** When on or err, the compiler checks for a prototype with a variable number of arguments, but the corresponding function definition specifies a fixed number of arguments. This will work in TopSpeed C, but it is a violation of the ANSI C rules for compatible function declarations, and therefore, not portable. The default setting is on.

---

**#pragma warn(wvnu => on | off | err)** **cp**

**Local variable never used.** When on or err, the compiler checks whether you declare a local variable and assign it a value but never use it. The default setting is on.

---

**#pragma warn(wovr => on | off | err)** **cp**

**Overflow in constant expression.** This warning is issued when a constant integer expression overflows. The default setting is on.

---

**#pragma warn(wacc => on | off | err)** **cp**

**Default access specifier used for base class.** This warning is issued if a base class specification does not have an access specifier and the default access is used (i.e. public for a struct and private for a class). The default setting is on.

---

**#pragma warn(wdel => on | off | err)** **cp**

**Expression in delete[] is obsolete.** This warning is issued if an expression is specified in the square brackets of a delete expression. The expression is ignored. This is obsolete C usage. The default setting is on.

---

**#pragma warn(wovl => on | off | err)** **cp**

**Keyword 'overload' is not required.** This warning is issued if the keyword overload is specified in C. The use of this keyword is obsolete C usage. The default setting is on.

---

**#pragma warn(wcic => on | off | err)** **cp**

**Constant in code segment requires initialization.** This warning is issued if a constant placed in the code segment requires run-time initialization, as may

be the case for an object declared `const` in C, whose initializer is an expression, when the `const_in_code` `#pragma` is set on. This situation will lead to a protection violation in OS/2 and Window 3 protected mode applications, so `const_in_code` should be set off if this warning is encountered. The default setting is on.

**`#pragma warn(wcrt => on | off | err)` \_\_\_\_\_ **cp****

### **Class definition as function return type, missing ‘;’ after ‘}’?**

This warning is issued if a class is defined in a function return type specification. Such a construct is legal, but unusual, and frequently results from omitting a semicolon between a class definition and the following function declaration. The default setting is on.

## **Project #pragmas**

---

A `#pragma` with the class name `project` is used to pass information from a compile to the Project System. The value of the `#pragma` should be a string, which is then stored in the object file for use by the Project System. Whenever an object file is added to the link list, the text specified using this `#pragma` is executed as a Project System command. For example, if a header file includes the line:

```
#pragma project("#set myflag=on")
```

then whenever a source file that includes this header file is included in a project, the Project System macro `myflag` will be set. This might be used for processing later in the project file.

This `#pragma` may only appear in source files, not in a project file.

### **Save/Restore #pragmas**

The `save` `#pragma` saves the entire `#pragma` state, so you can later restore it with a `restore` `#pragma`. The `save` and `restore` `#pragmas` work in a stack-like manner, thus allowing you to nest them. For example:

```
/*save the #pragma state and enable the interrupt convention*/
#pragma save
#pragma call(interrupt => on)
/* interrupt functions are specified here */
#pragma restore
```

There is no limit on the number of saves, except the amount of memory available. These `#pragmas` may be used in source files or in a project file.

## Link #pragmas

---

```
#pragma link(<filename> {,<filename> })
#pragma linkfirst(<filename>)
```

These #pragmas may be specified in a project file, in which case the nominated files are added immediately to the link list. In addition, the link #pragma may be specified in a C or C++ source file, in which case the nominated files will be added to the link list when an autocompile command is executed in the Project System, if any files already on the link list had this #pragma specified. For example:

```
#pragma link(file1.obj, file2.obj, file3.lib)
#pragma linkfirst(initexe.obj)
```

If no extension is given .obj is assumed. Files specified using #pragma link are added to the end of the link list (unless already present). A file specified using #pragma linkfirst is linked before the link list. Only one file may be specified for each link using #pragma linkfirst.

## Link\_Option #pragmas

---

#pragmas with the class name link\_option are used to specify linker options. These #pragmas may only occur in project files.

The following link\_option #pragmas are available:

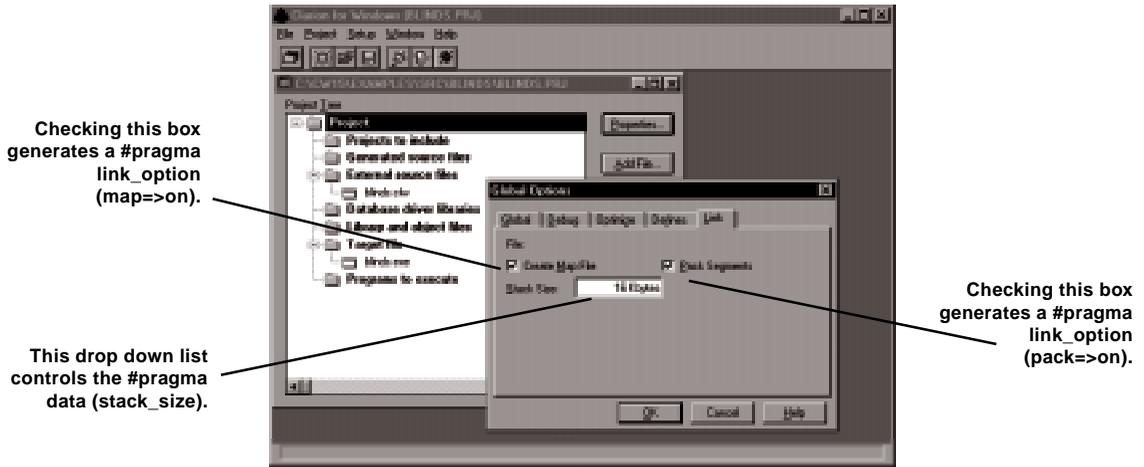
```
#pragma link_option(case => on | off)
#pragma link_option(decode => on | off)
#pragma link_option(link=> <string>)
#pragma link_option(map => on | off)
#pragma link_option(overlay => on | off)
#pragma link_option(pack => on | off)
#pragma link_option(shift => num)
#pragma link_option(share_const => on | off)
#pragma link_option(icon => iconname)
```

```
#pragma link_option(map => on | off) _____ p
```

Controls whether a map file is generated with information about segment sizes and publics etc. The default is to create a map file.

```
#pragma link_option(case => on | off) _____ p
```

This #pragma controls whether the linker treats upper/lower case as significant when linking. The default is case=>off.



**#pragma link\_option(pack => on | off ) p**

This #pragma controls whether segments are packed together. The default is pack=>on.

**#pragma link\_option(decode => on | off ) p**

This indicates whether the linker should produce deduced names in the MAP file, as well as their public symbols. The option is set to on if any C source files are included in a project, otherwise off.

**#pragma link\_option(shift => num) p**

This specifies the segment alignment shift count for new-format executables. The default is 4, indicating that segments are aligned on 16-byte boundaries.

**#pragma link\_option(link => <string> ) p**

This specifies the project system command to execute on #dolink.

**#pragma link\_option(share\_const=> on | off) p**

This pragma controls whether the 16-bit linker commons-up identical constants. The default is on, making the exe file smaller, but C programmers may want to turn it off if they are relying on constants having different addresses.

**#pragma link\_option(icon => iconname) p**

This pragma specifies the name of the application icon file (icon => MyIcon.ico).

## Define #pragmas

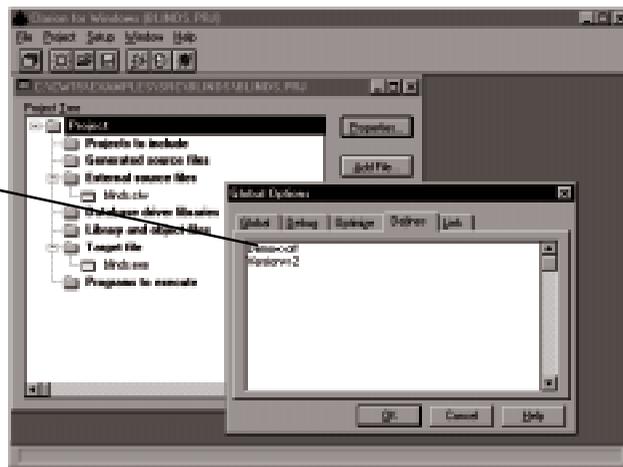
A #pragma whose class name is define is used to define a conditional compile symbol for subsequent compiles. The symbol is available for interrogation by the OMIT and COMPILE compiler directives. See the Language Reference for more information. This #pragma may only be used in project files.

A define #pragma takes the form:

```
#pragma define(ident=>value)
```

where *ident* names the symbol and *value* specifies the value it is given.

This entry generates a #pragma define (demo=>off), and a #pragma define (version=>2).



For Modula-2, the given identifier is defined as a boolean constant with value TRUE if the value *on* was specified, otherwise FALSE. For C and C++, the given identifier is defined as a macro. If the value *on* is specified, the macro is defined to the value 1. If the value *off* is specified, the macro is not defined. Any other value will cause the identifier to be defined as a macro expanding to the given value. Only a single C or C++ token may be specified, or the compiler will report an error. To define a macro where the value is a string literal, use a command of the form #pragma define (name => “fred”).

**#pragma define(maincode => on | off ) \_\_\_\_\_ p**

Enables (on—the default) or disables (off) generation of initialization code. Turn maincode off when compiling generic modules or LIB modules.

**#pragma define(zero\_divide => on | off ) \_\_\_\_\_ p**

Specifies divide by zero behavior. When on, division by zero returns zero. When off (the default), division by zero returns an exception.

**#pragma define(logical\_round => on | off ) **p****

Specifies rounding behavior when truncating a REAL to a LONG. When on, the result is rounded up if the REAL value is “close to” the next larger integer. When off (default), no rounding occurs.

**#pragma define(stack\_threshold => size ) **p****

Specifies the size (in bytes) of the threshold at which any data structure larger than the specified size (default is 16384) is assigned heap memory instead of stack memory.

**#pragma define(BCD\_Arithmetic => on | off ) **p****

Specifies use of Binary Coded Decimal (BCD) arithmetic when on (default) and forces use of Floating Point arithmetic when off.

**#pragma define(BCD\_ULONG => on | off ) **p****

Specifies use of Binary Coded Decimal (BCD) arithmetic for ULONG variables when on (default) and forces use of Floating Point when off.

**#pragma define(BCD\_Large => on | off ) **p****

Enables or disables use of DECIMAL and PDECIMAL variables greater than fifteen (15) digits. The default is on (enabled).

**#pragma define(big\_code => n) **p****

Specifies number of procedures per segment. By default (n = 0) all procedures in a single module go into a single code segment. Setting the value of n for a specific module “breaks up” the module containing a large number of procedures which “breaks” the 64K code segment limit.

**#pragma define(profile => on | off ) **p****

Specifies the compiler will invoke a procedure call at the beginning and end of compiling each procedure. This allows you to implement your own profiler. The prototypes for these procedures must be:

```
EnterProc(UNSIGNED Line,*CSTRING Proc,*CSTRING,
File),NAME('Profile:EnterProc)
LeaveProc(),NAME('Profile:LeaveProc)
```

The EnterProc is called at the beginning of each procedure and LeaveProc at the end.

**#pragma define(init\_priority => n) **p****

Specifies a number (n) which is compatible with the C++ module priority schema. Default is 5.

## Pre-defined Flags

---

Whenever you #compile a program the project system automatically defines a number of flags to ON or OFF, depending on the target system. You may use these predefined flags to control your make process. There are four that the compiler automatically creates for you that you can use in OMIT and COMPILE statements for conditional compilation.

|                  |                                                |
|------------------|------------------------------------------------|
| <b>_WIDTH32_</b> | On for 32-bit applications and off for 16-bit. |
| <b>_CW_</b>      | On for Clarion.                                |
| <b>_CLW20_</b>   | On for Clarion, version 2.0                    |
| <b>_CDD_</b>     | On for Clarion for DOS.                        |

## Project System Examples

Following is an example of some project system commands that we use here at TopSpeed to make our file drivers. This example uses a wide variety of project system statements and shows how the project system can be used to control the accuracy and completeness of even the most complicated projects.

These example statements are divided among four files, showing the project system's ability to support structured programming, modularity, and reusability. The files are ALLDRV.PR, which #calls ORACLE.PR (among others), which #includes SQLFILES.PR, which in turn #includes DRVKIT.PI.

### ALLDRV.PR

```
#system win dll
#model clarion

#set drvdebug = full
#set kitdebug = full

#set release = off
#set fromclw = on
#set incbuildno = off
#set demo = off

#if "%release"="on" #then
 #pragma define(_RELEASE=>on)
 #set incbuildno = on
 #set kitdebug = off
 #set drvdebug = off
#endif

#pragma define(DEMO_VERSION=>%demo)

#set domodels=
.
#abort on
#set down32=off #set dolib=off
#call %prjfile
#set down32=off #set dolib=on
#call %prjfile
#set down32=off #set dolib=off
#abort off
#set down32=on #set dolib=off
#call %prjfile
#set down32=on #set dolib=on
#call %prjfile
#abort on
.

#if #exists btrieve.pr #then #set prjfile=btrieve.pr %domodels #endif
#if #exists odbc.pr #then #set prjfile=odbc.pr %domodels #endif
#if #exists cla21.pr #then #set prjfile=cla21.pr %domodels #endif
```

```

#if #exists tps.pr #then #set prjfile=tps.pr %domodels #endif
#if #exists dos.pr #then #set prjfile=dos.pr %domodels #endif
#if #exists ascii.pr #then #set prjfile=ascii.pr %domodels #endif
#if #exists basic.pr #then #set prjfile=basic.pr %domodels #endif

#set domodels=
'
#set down32=off #set dolib=off
#call %prjfile
#set down32=off #set dolib=on
#call %prjfile
#set down32=off #set dolib=off
'

#file redirect ts.red
#if #exists sql400.pr #then #set prjfile=sql400.pr %domodels #endif
#if #exists oracle.pr #then #set prjfile=oracle.pr %domodels #endif

```

## **ORACLE.PR:**

```

#noedit

-- ORACLE.PR Oracle Driver project file

#system win dll --target OS is windows, dll executable
#model clarion --memory model is clarion

-- Set default macro values. These "switches" will control the make
-- process
#set drv = ORA
#set trace = off
#set heapchk = off
#set drvdebug = full
#set sqldebug = full
#set kitdebug = off
--#set release = on

#expand ORACLEIN.CPP --set %cpath to path where file is created
-- %opath to path where file is opened
-- %ext to CPP
-- %tail to ORACLEIN
-- %cdir to directory where file is created
-- %odir to directory where file is opened

#set drvdir = %odir --save the %odir value
#set sql_type = 0 --set %sql_type to 0

-- Define a conditional compile symbol for subsequent compiles.
-- The symbol is DRVSPEC, and its value is "oraclesp.h"
-- DRVSPEC is available for interrogation by the OMIT and COMPILER
-- statements—see the Language Reference for more information.
#pragma define(DRVSPEC=>'oraclesp.h')

--Compile the sql modules with appropriate levels of debug code.
#include SQLFILES.PR --Execute statements from SQLFILES.PR here.

```

```

--All macros, pragmas, and link list are fully
--available to the #included statements.

#compile ORACLEIN.CPP --compile the oracle c++ source.
#compile ORAIMPOR.CLW /define (maincode=>off) --and the clarion source.
 --both are added to the link list.
#pragma link(ORAIMPOR.RSC) --add ORAIMPOR.RSC to the link list.
#pragma link (ORA7WIN.LIB) --add ORA7WIN.LIB to the link list.
#pragma link (%lnkpfx%asc%S%.LIB) --add CWASC16.LIB to the link list.
 --%lnkpfx% resolves to CW,
 --%S% resolves to "16"

-- Execute the series of statements assigned to drv_Link at the very
-- end of the DRVKIT.PI file. These statements are designed to link
-- and patch the File Driver.
%drv_Link

```

### **SQLFILES.PR:**

```

-- Release version: Disable all debugging and tracing
#if "%release"="on" #then
 #set drvdebug = off
 #set kitdebug = off
 #set sqldebug = off
 #set trace = off
 #set heapchk = off
#endif

-- make sure the sql_type switch is explicitly set (no default)
#if '%sql_type' = '' #then
 #error "sql_type must be set"
#endif

-- Default is compact code. Set the DRIVER_COMPACT symbol based on
-- the value of the %compact macro
#if "%compact"="" #then #set compact=on #endif
#if '%compact'='off' #or '%heapchk'='on' #then
 #pragma define(DRIVER_COMPACT=>off)
#else
 #pragma define(DRIVER_COMPACT=>on)
#endif

#include DRVKIT.PI --Execute statements from DRVKIT.PI here.
 --All macros, pragmas, and link list are fully
 --available to the #included statements.

#pragma save --Save the current #pragma settings so they
 --can be restored later with #pragma restore.

-- Debugging: Debugger info and Run-time checks
#if "%sqldebug"="" #then #set sqldebug=off #endif --default is off
#pragma debug(vid=>%sqldebug)
#if "%sqldebug"="full" #then --for full debugging, enable
 #pragma check(index=>on,range=>on,overflow=>on) --runtime error checks
 #pragma debug(line_num=>on) --and line numbering

```

```

#endif

#pragma warn(wall=>on) --enable all warning msgs

#set srcfile = SAFESTR.CPP --set %srcfile to SAFESTR.CPP
#set dstfile = %sql_type%SAFE.CPP --set %dstfile to OSAFE.CPP
-- Execute the series of statements assigned to makesrc in the middle
-- of the DRVKIT.PI file. These statements are designed to get the C++
-- constructor entry point to have a different name for each driver.
%makesrc

#set srcfile = SQLOPEN.CPP -- ditto
#set dstfile = %sql_type%SQLOPE.CPP
%makesrc

#set srcfile = SQLUPDAT.CPP -- ditto
#set dstfile = %sql_type%SQLUPD.CPP
%makesrc

#set srcfile = SQLRETRI.CPP -- ditto
#set dstfile = %sql_type%SQLRET.CPP
%makesrc

#set srcfile = SQLGLOB.CPP -- ditto
#set dstfile = %sql_type%SQLGLO.CPP
%makesrc

#if %system=win #then -- conditionally
#set srcfile = SQLVIEW.CPP -- ditto
#set dstfile = %sql_type%SQLVIEW.CPP
%makesrc
#endif

#pragma restore --restore the #pragma settings saved earlier
#pragma warn(wall=>on) --enable all warning messages

```

## **DRVKIT.PI:**

```

#noedit

-- DRVKIT.PI Driver Kit project include file

-- DrvKit.Pi contains project statements common to all Driver Kit based
-- File Drivers. The following settings must be set before including
-- drvkit.pi:
--
-- #set drv = A 3 or 4 letter driver id (e.g C21)
-- #pragma define(DRVSPEC="c21specs.h") -- or appropriate file
--
-- Any other defines that affect the Driver Specification
--
-- Optional:
--

```

```

-- #set trace = on -- Enable tracing
-- #set heapchk = on -- Enable Heap Checker
-- #set common = on -- Merge common code
--
-- Release version: Disable all debugging and tracing
#if "%release"="on" #then
 #set drvdebug = off
 #set kitdebug = off
 #set trace = off
 #set heapchk = off
#endif

-- Windows: Ensure Clarion 4 Windows conventions are adopted
#pragma warn(wall=>on) --enable all warning messages
#pragma define(__CLARION__=>on) --set compiler directive switch

#if #not %system=dos #then -- dos,
 #if "%dowin32"=on #then
 #system win32 dll -- win32, or
 #else
 #system win dll -- win16
 #endif
 #model clarion

 #if %filetype=dll #then
 #pragma define(_WINDLL=>on) --set compiler directive switch
 #endif

-- Build the appropriate file names for this driver set.
#if "%dolib"="" #then
 #set dolib=off --default dolib = off
#endif
#if "%dowin32"="" #then
 #set dowin32=off --default dowin32 = off
#endif
#if %dolib=on #then --set link (.lib) prefix
 #set lnkpfx = CL
#else
 #set lnkpfx = CW
#endif
#if "%pfx"="" #then
 #set pfx = %drv% --set prefix to driver name
#endif
#else
 #set dolib=off
 #if %model=extendll #then
 #pragma define(_XTDDL=>on)
 #endif

 #set pfx = %%%drv%
 #if '%RWMODE%' = 'on' #then
 #set lnkpfx=drw
 #else
 #set lnkpfx=%clapfx%
 #endif
 #set S = "" --set suffix to null
#endif

```

```

#set drvname = %lnkpfx%drv%S% --Put the name together
#message "Making %drvname% File Driver" --Display status message
#if #not "%inbrowser" #then
 #if #exists %drvname%.ver #then --Conditionally...
 #compile %drvname%.ver -- compile the driver
 #endif
#endif

-- Heap Checker: Compile and enable Heap Checker (No Debugging)
#if "%heapchk"="on" #for "%heapdbg"="on" #then
#set heapchk = on
#pragma define(HEAPCHK=>on) --Set compiler directive switch.
#if "%heapdll"="on" #then --If dll, then
 #pragma link(%clapfx%hchk.lib) --add heapchk lib to link list.
#else
 #pragma save --Save current pragma settings.
 #if "%heapdbg"="on" #then --Conditionally...
 #pragma debug(vid=>full) -- enable debug code.
 #endif
 #compile HEAPCHK.C #to %pfx%HCHK.OBJ --Compile heap checker.
 #compile STRCHK.C #to %pfx%SCHK.OBJ --
 #compile NEW.CPP #to %pfx%NEW.OBJ
 #pragma restore --Restore saved pragma settings.
#endif
#endif

-- Debugging: Debugger info and Run-time checks
#if "%drvdebug"="" #then #set drvdebug=full #endif--default is "full"
#pragma debug(vid=>%drvdebug)
#if "%drvdebug"="full" #then
 #pragma check(index=>on,range=>on,overflow=>on) --enable runtimes
 #pragma debug(line_num=>on) --enable line nos.
#endif

-- Common Code: Some Driver Kit code is merged when linking multiple
-- File Driver Libraries
#if "%common"="on" #then --Conditionally...
 #pragma define(COMMON_CODE=>on) -- define compiler switch
#endif

#if '%drvdir' = '' #then
 #error "drvdir must be set to build lib versions of the drivers"
#endif

-- To get the C++ constructor entry point to have a different
-- name for each of the drivers it is necessary to compile
-- different C++ source modules:
#set makesrc = '

```

```

 #if (#not #exists %drvdir%%dstfile) #for (%drvdir%%dstfile #older
 %%srcfile) #for (%srcfile #older %drvdir%%dstfile) #then
 #expand %srcfile
 #run "copy %opath %drvdir%%dstfile > NUL "
 #endif
 #compile %%dstfile %%defns
,

-- Driver Kit: Compile Driver Kit sources
#pragma save --First, save current pragma settings
#if #not ""kitdebug"" #then --Conditionally...
 #pragma debug(vid=>%kitdebug) -- set debug level
#else
 #pragma debug(vid=>off)
#endif

#set srcfile = DRVL1.C --Set srcfile name
#set dstfile = %pfx%L1.C --set dstfile name w correct prefix
%makesrc --Execute stmts defined above.

#if #not "%nocommon"="on" #then--if common code
#set srcfile = DRVSTATE.C --make DRVSTATE
#set dstfile = %pfx%STAT.C
%makesrc

#if #not (%system=dos) #then --if system is not DOS
#set srcfile = DRVVIEW.CPP --make DRVVIEW
#set dstfile = %pfx%VIEW.CPP
%makesrc

#if #not (%filetype=dll) #then --Conditionally...
 #pragma define(DRV_HAS_LIBMAIN=>on) -- set compiler switch
#endif

#set srcfile = DRW.C
#set dstfile = %pfx%W%dolib%.C
#set defns = '/define(_LIB_TARGET=>%dolib%)'
%makesrc
#set defns = ''

#set srcfile = DRWUTIL.C --make DRWUTIL
#set dstfile = %pfx%WUTI.C
%makesrc

#if #not %dolib% #then --Conditionally
#set srcfile = DRVDIAL.CLW -- make DRVDIAL
#set dstfile = %pfx%DIAL.CLW
#set defns = '/define(maincode=>off)'
%makesrc
#set defns = ''
#pragma link(%pfx%dial.rsc) --Add DIAL to link list
#endif
#endif
#endif
#if "%trace"="on" #then --Conditionally...
#pragma save, define	TRACE=>on) -- save settings...
#set srcfile = DRVTRACE.C -- make DRVTRACE
#set dstfile = %pfx%TRAC.C
%makesrc

```

```

#endif

#set srcfile = DRVPIPE.C --make DRVPIPE
#set dstfile = %pfx%P%dolib%.C
#set defns = '/define(_LIB_TARGET=>%dolib%)'
#makesrc
#set defns = ''

#if "%trace"="on" #then
 #pragma restore --restore saved pragma settings
#endif
#pragma restore

-- Build Macro drv_Link to be used later in this process
-- to link and patch the File Driver:
#set drv_Link =
'

#pragma link_option(share_const=>on)

#if #not (%%system=dos) #then
 #if "%dolib"="on" #then
 #dmlink %%drvname%.lib
 #else
 #implib %%drvname%.lib %%drvname%.exp
 #if define(_CW15)=on #then
 #pragma linkfirst(idll%%S%%w.obj)
 #else
 #pragma linkfirst(icwdll.obj)
 #endif
 #pragma link(win%%S%.lib)
 #pragma link(cwrun%%S%.lib)
 #pragma link_option(decode=>off)
 #dmlink %%drvname%.dll
 #endif

 #if "%make" #and #not "%dolib"="on" #then
 #exemod %%drvname%.dll %%drvname%.exp %%drvname%.map
 #endif
#else
 #if %%filetype=dll #then
 #implib %%drvname%.lib %%drvname%.exp
 #endif
 #set tscla = on
 #set tscpp = off

 #link %%drvname%

 #if "%make" #and (%%filetype=dll) #then
 #expand %%drvname%.dll
 #run "mkdriver %%cpath > NUL"
 #endif
#endif
'

```

## Module Definition Files (.EXP Files)

A module definition file describes the name, attributes, exports, and other characteristics of a dynamic-link library for Microsoft Windows. This file is required for Windows.

A module definition file (.EXP) is generated whenever you make a new project, or a project whose target type, operating system, or run-time library has changed.

### 16-Bit

---

#### Module Definition File Syntax

A module definition file contains one or more statements. Each statement defines an attribute of the executable file, such as its module name, the attributes of program segments, and the numbers and names of exported symbols. The statements and the attributes they define are listed below:

| <b>Statement</b> | <b>Attribute</b>                           |
|------------------|--------------------------------------------|
| NAME             | Names application                          |
| LIBRARY          | Names dynamic-link library                 |
| CODE             | Gives default attributes for CODE segments |
| DATA             | Gives default attributes for DATA segments |
| SEGMENTS         | Gives attributes for specific segments     |
| STACKSIZE        | Specifies local stack size in bytes        |
| EXPORTS          | Defines exported functions                 |
| HEAPSIZE         | Specifies local heap size                  |
| EXETYPE          | Identifies operating system                |

The following rules govern the use of these statements:

- ◆ If you use either a NAME or a LIBRARY statement, it must precede all other statements in the module definition file.
- ◆ You can include source-level comments in the module definition file, by beginning a line with a semicolon(;). The utilities ignore each such comment line.
- ◆ Module definition keywords (such as NAME, LIBRARY, and SEGMENTS) must be entered in uppercase letters.

## Example—Module Definition File

The following example gives module definitions for a dynamic-link library:

```
LIBRARY MyDLL
; Sample export file
EXPORTS
 Func1 @1
 Var1 @2
 Func2 @3
 Func3 @4
 Func4 @5
```

## The NAME Statement

The NAME statement identifies the file as an executable application (rather than a DLL) and optionally defines the name and application type.

**NAME [appname] [apptype]**

|                |                                                                                                                                                                                                                               |
|----------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>appname</i> | If appname is given, it becomes the name of the application as it is known by the operating system. If no appname is given, the name of the executable file, with the extension removed, becomes the name of the application. |
| <i>apptype</i> | Used to control the program's behavior under Windows. This information is kept in the executable-file header. The apptype field may have one of the following values:                                                         |
| WINDOWAPI      | The application uses the API provided by Windows and must be executed in the Windows environment.                                                                                                                             |
| GUI            | Same as WINDOWAPI.                                                                                                                                                                                                            |
| CUI            | The program uses a character based user interface, like DOS.                                                                                                                                                                  |

If the NAME statement is included in the module-definition file, then the LIBRARY statement cannot appear.

If neither a NAME statement nor a LIBRARY statement appears in a module-definition file, NAME is assumed.

The following example assigns the name wdemo to the application being defined:

```
NAME wdemo GUI
```

## The LIBRARY Statement

The LIBRARY statement identifies the file as a dynamic-link library. The name of the library, and the type of library module initialization required, may also be specified.

### **LIBRARY [libraryname][initialization]**

*libraryname* If libraryname is specified, it becomes the name of the library as it is known by the operating system. This name can be any valid file name. If no libraryname is given, the name of the executable file, with the extension removed, becomes the name of the library.

*initialization* The initialization field is optional and can have one of the two values listed below. If neither is given, then the initialization default is INITINSTANCE.

**INITGLOBAL** The library-initialization routine is called only when the library module is initially loaded into memory.

**INITINSTANCE** The library-initialization routine is called each time a new process gains access to the library.

If the LIBRARY statement is included in a module definition file, then the NAME statement cannot appear.

The following example assigns the name mydll to the dynamic-link module being defined, and specifies that library initialization is performed each time a new process gains access to myDLL:

```
LIBRARY myDLL INITINSTANCE
```

## The CODE Statement

The CODE statement defines the default attributes for CODE segments within the application or library.

### **CODE [attribute<sub>1</sub>] [attribute<sub>2</sub>] ... [attribute<sub>n</sub>]**

*attribute* Each attribute specified must correspond to one of the following attribute fields. Each field can appear at most one time, and order is not significant. The attribute fields are presented below, along with legal values. In each case, the default value is listed last.

| Field              | Alternative Values |               |
|--------------------|--------------------|---------------|
| <i>load</i>        | PRELOAD            | LOADONCALL    |
| <i>executeonly</i> | EXECUTEONLY        | EXECUTEREAD   |
| <i>iopl</i>        | IOPL               | NOIOPL        |
| <i>conforming</i>  | CONFORMING         | NONCONFORMING |
| <i>shared</i>      | SHARED             | NONSHARED     |
| <i>moveable</i>    | MOVEABLE           | FIXED         |
| <i>discard</i>     | NONDISCARDABLE     | DISCARDABLE   |

The *load* field determines when a code segment is to be loaded. This field contains one of the following keywords:

- PRELOAD**      The segment is loaded automatically, at the beginning of the program, and, when using the TopSpeed Overlay System, will not be swapped out.
- LOADONCALL**    The segment is not loaded until accessed (default).

The *executeonly* field determines whether a code segment can be read as well as executed. This field contains one of the following keywords:

- EXECUTEONLY**    The segment can only be executed.
- EXECUTEREAD**    The segment can be both executed and read (default).

The *iopl* field determines whether or not a segment has I/O privilege (that is, whether it can access the hardware directly). This field contains one of the following keywords:

- IOPL**            The CODE segment has I/O privilege.
- NOIOPL**          The CODE segment does not have I/O privilege (default).

The *conforming* field specifies whether or not a code segment is a 286 conforming segment. The concept of a conforming segment deals with privilege level (the range of instructions that the process can execute) and is relevant only to users writing device drivers and system-level code. A conforming segment can be called from either Ring 2 or Ring 3, and the segment executes at the caller's privilege level. This field contains one of the following keywords:

- CONFORMING**    Indicates a 286 conforming code segment.
- NONCONFORMING**  
                      Indicates a non-conforming code segment (default).

The *shared* field determines whether all instances of the program can share a given code segment. This field is provided for use with real-mode Windows. The *shared* field contains one of the following keywords:

|           |                                                                     |
|-----------|---------------------------------------------------------------------|
| SHARED    | All instances of the program can share code segments.               |
| NONSHARED | Code segments cannot be shared between program instances (default). |

The *moveable* field determines whether a segment can be moved around in memory. This field is provided for use with real-mode Windows. The *moveable* field contains one of the following keywords:

|          |                                              |
|----------|----------------------------------------------|
| MOVEABLE | Code segments can be relocated in memory.    |
| FIXED    | Code segments cannot be relocated (default). |

The *discard* field determines whether a segment can be swapped out to disk by the operating system when not currently needed. This attribute is provided for use with real-mode Windows. The *discard* field contains one of the following keywords:

|                |                                                |
|----------------|------------------------------------------------|
| DISCARDABLE    | Code segments can be swapped out to disk.      |
| NONDISCARDABLE | Code segments cannot be swapped out (default). |

The following example sets defaults for the module's code segments, so that they are not loaded until accessed and so that they have I/O hardware privilege:

```
CODE LOADONCALL IOPL
```

### **The DATA Statement**

The DATA statement defines the default attributes for the DATA segments within the application or library.

**DATA [attribute<sub>1</sub>] [attribute<sub>2</sub>] ... [attribute<sub>n</sub>]**

*attribute* Each attribute must correspond to one of the following attribute fields. Each field can appear at most once, and order is not significant. The attribute fields are listed below, along with the legal values for each field. In each case, the default value is listed last.

| Field           | Alternative Values |                   |
|-----------------|--------------------|-------------------|
| <i>load</i>     | PRELOAD            | LOADONCALL        |
| <i>readonly</i> | READONLY           | READWRITE         |
| <i>instance</i> | NONE               | SINGLE   MULTIPLE |
| <i>shared</i>   | SHARED             | NONSHARED         |
| <i>moveable</i> | MOVEABLE           | FIXED             |
| <i>discard</i>  | DISCARDABLE        | NONDISCARDABLE    |

The *load* field determines when a segment will be loaded. This field contains one of the following keywords:

|            |                                                                                                                               |
|------------|-------------------------------------------------------------------------------------------------------------------------------|
| PRELOAD    | The segment is loaded when the program begins execution and, when using the TopSpeed Overlay System, will not be swapped out. |
| LOADONCALL | The segment is not loaded until it is accessed (default).                                                                     |

The *readonly* field determines the access rights to a DATA segment. This field contains one of the following keywords:

|           |                                                        |
|-----------|--------------------------------------------------------|
| READONLY  | The segment can only be read.                          |
| READWRITE | The segment can be both read and written to (default). |

The *instance* field affects the sharing attributes of the automatic DATA segment, which is the physical segment represented by the group name DGROUP. This segment group makes up the physical segment which contains the local stack and heap of the application. This field contains one of the following keywords:

|          |                                                                                                                                                                                         |
|----------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| NONE     | No automatic DATA segment is created.                                                                                                                                                   |
| SINGLE   | A single automatic DATA segment is shared by all instances of the module. In this case, the module is said to have “solo” data. This keyword is the default for dynamic-link libraries. |
| MULTIPLE | The automatic DATA segment is copied for each instance of the module. In this case, the module is said to have “instance” data. This keyword is the default for applications.           |

The *shared* field determines whether all instances of the program can share a READWRITE DATA segment.

|        |                                                                                                  |
|--------|--------------------------------------------------------------------------------------------------|
| SHARED | One copy of the DATA segment will be loaded and shared among all processes accessing the module. |
|--------|--------------------------------------------------------------------------------------------------|

**NONSHARED**      The segment cannot be shared, and must be loaded separately for each process (default).

The *moveable* field determines whether a segment can be moved around in memory. This field is provided for use with real-mode Windows. This field contains one of the following keywords:

**MOVABLE**              Data segments can be relocated in memory.

**FIXED**                  Data segments cannot be relocated (default).

The optional *discard* field determines whether a segment can be swapped out to disk by the operating system, when not currently needed. This attribute is provided for use with real-mode Windows and the TopSpeed Overlay System. This field contains one of the following keywords:

**DISCARDABLE**      Data segments can be swapped out to disk.

**NONDISCARDABLE**  
Data segments cannot be swapped out (default).

**Note:**    **Care should be taken not to specify contradictory segment attributes.**

The following example defines the default attributes for DATA segments so that they are loaded only when accessed, and cannot be shared by more than one copy of the program:

```
DATA LOADONCALL NONSHARED
```

By default, the DATA segment can be read and written, and the automatic DATA segment is copied for each instance of the module.

### **The SEGMENTS Statement**

The SEGMENTS statement defines the attributes of one or more individual segments in the application or library on a segment-by-segment basis. The attributes specified by this statement override defaults set in CODE and DATA statements.

```
SEGMENTS
 segmentdefinitions
```

The **SEGMENTS** keyword marks the beginning of the segment definitions. This keyword can be followed by one or more segment definitions, each on a separate line. The syntax for each segment definition is as follows:

**segname [CLASS 'classname'] [attribute<sub>1</sub>] [attribute<sub>2</sub>] ... [attribute<sub>n</sub>]**

|                  |                                                                                                                                                                                                                                    |
|------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>segname</i>   | Begins the segment definition. The segname can be placed in optional single quotation marks (''). The quotation marks are required if the segname conflicts with a module definition keyword, such as <b>CODE</b> or <b>DATA</b> . |
| <b>CLASS</b>     | Specifies the class of the segment. The single quotation marks ('') are required around classname. If <b>CLASS</b> is not specified, it is assumed to be <b>CODE</b> .                                                             |
| <i>attribute</i> | Must correspond to one of the attribute fields described for the <b>CODE</b> and <b>DATA</b> statements above. Each field can appear at most once, and order is not significant.                                                   |

The following example specifies segments named **pr1\_TEXT**, **pr2\_TEXT** and **pr3\_TEXT**. Each segment is given different attributes.

```
SEGMENTS
 pr1_TEXT IOPL
 pr2_TEXT EXECUTEONLY PRELOAD
 pr3_TEXT LOADONCALL READONLY
```

### The STACKSIZE Statement

The **STACKSIZE** statement defines the size of the application's stack, in bytes.

**STACKSIZE number**

|               |                                                                                                                                                                                         |
|---------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>number</i> | Specifies stack size in bytes. Number must be an integer and is considered to be in decimal format by default, but you can use C notation to specify hexadecimal or octal. For example: |
|---------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

```
STACKSIZE 4096
```

### The HEAPSIZE Statement

The **HEAPSIZE** statement defines the size of the application's local heap, in bytes. This value affects the size of the automatic **DATA** segment.

**HEAPSIZE number***number*

Specifies heap size in bytes. Number must be an integer and is considered to be in decimal format by default, but you can use C notation to specify hexadecimal or octal. For example:

```
HEAPSIZE 4000
```

**The EXETYPE Statement**

The EXETYPE statement specifies in which operating system the application (or dynamic-link library) is to run. This statement is optional and provides an additional degree of protection against the program being run in an incorrect operating system.

**EXETYPE [WINDOWS | DOS4]**

The effect of EXETYPE is simply to set bits in the header which identify operating system type. The operating system loaders may or may not check these bits. When writing programs for Microsoft Windows, EXETYPE WINDOWS must be specified.

**The EXPORTS Statement**

The EXPORTS statement defines the names and attributes of the functions exported to other modules, and of the functions that run with I/O privilege. The term “export” refers to the process of making a function available to other run-time modules. By default, functions are hidden from other modules at run time.

**EXPORTS  
exportdefinitions**

The EXPORTS keyword marks the beginning of the export definitions. It may be followed by up to 3072 export definitions, each on a separate line. You should give an export definition for each dynamic-link routine that you want to make available to other modules. The syntax for an export definition is as follows:

**entryname [pwords] @number | ? [NODATA]***entryname*

Defines the function name as it is known to other modules.

|               |                                                                                                                                                                                                                                                                                                                                                                                          |
|---------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>pwords</i> | Specifies the total size of the function's parameters, as measured in words (the total number of bytes divided by two). This field is required only if the function executes with I/O privilege. When a function with I/O privilege is called, OS/2 consults the <i>pwords</i> field to determine how many words to copy from the caller's stack to the I/O-privileged function's stack. |
| @number   ?   | Defines the function's ordinal position within the module-definition table. The @ may be followed by the position number of the function, or it may be followed by a question mark ( ? ) if the position is unknown. The numbers must be in sequence.                                                                                                                                    |
| NODATA        | Provided for use by real-mode Windows (optional).                                                                                                                                                                                                                                                                                                                                        |

The EXPORTS statement is meaningful for functions within dynamic link libraries, functions which execute with I/O privilege, and call back functions in Windows programs.

For example:

```
EXPORTS
 Func1 @?
 Func2 @?
 CharTest @?
```

## Exporting CLASSES

Exporting CLASS declarations requires a special form of export definition.

You must create two export definitions for the CLASS itself. The first begins with VMT\$ followed by the name of the CLASS as the *entryname*. The second begins with TYPE\$ followed by the name of the CLASS as the *entryname*. These are followed by an export definition for each method in the CLASS to export whose *pwords* must begin with the name of the CLASS as the first parameter.

For example:

```
EXPORTS
 VMT$MYCLASS @?
 TYPE$MYCLASS @?
 FIRSTMETHOD@F7MYCLASS @?
 FIRSTMETHOD@F7MYCLASS @?
```

## 32-Bit

---

### Module Definition File Syntax

A module definition file contains one or more statements. Each statement defines an attribute of the executable file, such as its module name, the attributes of program segments, and the numbers and names of exported symbols. The statements and the attributes they define are listed below:

| <b>Statement</b>  | <b>Attribute</b>                |
|-------------------|---------------------------------|
| NAME              | Names the application           |
| LIBRARY           | Names the dynamic-link library  |
| HEAP_COMMIT       | Amount of heap committed        |
| HEAP_RESERVE      | Amount of heap reserved         |
| STACK_COMMIT      | Amount of stack committed       |
| STACK_RESERVE     | Amount of stack reserved        |
| IMAGE_BASE        | Module base memory location     |
| DEBUG             | Include debug information       |
| LINENUMBERS       | Include line number information |
| SECTION_ALIGNMENT | Multiples of 4096 only          |
| FILE_ALIGNMENT    | Multiples of 512 only           |
| EXPORTS           | Defines exported functions      |

The following rules govern the use of these statements:

- ◆ If you use either a NAME or a LIBRARY statement, it must precede all other statements in the module definition file.
- ◆ You can include source-level comments in the module definition file, by beginning a line with a semicolon(;). The utilities ignore each such comment line.
- ◆ Module definition keywords (such as NAME, LIBRARY, and EXPORTS) must be entered in uppercase letters.
- ◆ The EXPORTS statement must appear last.

### Example—Module Definition File

The following example gives module definitions for a dynamic-link library:

```

LIBRARY MyDLL
; Sample export file
EXPORTS
 Func1 @1
 Var1 @2
 Func2 @3
 Func3 @4
 Func4 @5

```

## **The NAME Statement**

The NAME statement identifies the file as an executable application (rather than a DLL) and optionally defines the name and application type.

**NAME [appname] [apptype]**

|                |                                                                                                                                                                                                                               |
|----------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>appname</i> | If appname is given, it becomes the name of the application as it is known by the operating system. If no appname is given, the name of the executable file, with the extension removed, becomes the name of the application. |
| <i>apptype</i> | Used to control the program's behavior under Windows. This information is kept in the executable-file header. The apptype field may have one of the following values:                                                         |
| WINDOWAPI      | The application uses the API provided by Windows and must be executed in the Windows environment.                                                                                                                             |
| GUI            | Same as WINDOWAPI.                                                                                                                                                                                                            |
| CUI            | The program uses a character based user interface, like DOS.                                                                                                                                                                  |

If the NAME statement is included in the module-definition file, then the LIBRARY statement cannot appear.

If neither a NAME statement nor a LIBRARY statement appears in a module-definition file, NAME is assumed.

The following example assigns the name wdemo to the application being defined:

```
NAME wdemo WINDOWAPI
```

## The LIBRARY Statement

The LIBRARY statement identifies the file as a dynamic-link library. The name of the library, and the type of library module initialization required, may also be specified.

**LIBRARY [libraryname][initialization]**

*libraryname* If libraryname is specified, it becomes the name of the library as it is known by the operating system. This name can be any valid file name. If no libraryname is given, the name of the executable file, with the extension removed, becomes the name of the library.

*initialization* The initialization field is optional and can have one of the two values listed below. If neither is given, then the initialization default is INITINSTANCE.

**INITGLOBAL** The library-initialization routine is called only when the library module is initially loaded into memory.

**INITINSTANCE** The library-initialization routine is called each time a new process gains access to the library.

If the LIBRARY statement is included in a module definition file, then the NAME statement cannot appear.

The following example assigns the name mydll to the dynamic-link module being defined, and specifies that library initialization is performed each time a new process gains access to myDLL:

```
LIBRARY myDLL INITINSTANCE
```

## The HEAP\_COMMIT Statement

Specifies the amount of heap committed. This statement is not generated from the Clarion environment, but may be specified by manually editing the .EXP file and running the linker standalone. The syntax for the HEAP\_COMMIT statement is as follows:

**HEAP\_COMMIT number**

## The STACK\_COMMIT Statement

Specifies the amount of stack committed. This statement is not generated from the Clarion environment, but may be specified by manually editing the .EXP file and running the linker standalone. The syntax for the STACK\_COMMIT statement is as follows:

**STACK\_COMMIT number**

### **The HEAP\_RESEERVE Statement**

Specifies the amount of heap reserved. This statement is not generated from the Clarion environment, but may be specified by manually editing the .EXP file and running the linker standalone. The syntax for the HEAP\_RESERVE statement is as follows:

**HEAP\_RESERVE number**

### **The STACK\_RESERVE Statement**

Specifies the amount of stack reserved. This statement is not generated from the Clarion environment, but may be specified by manually editing the .EXP file and running the linker standalone. The syntax for the STACK\_RESERVE statement is as follows:

**STACK\_RESERVE number**

### **The IMAGE\_BASE Statement**

Specifies the base memory location of the module. This statement is not generated from the Clarion environment, but may be specified by manually editing the .EXP file and running the linker standalone. The syntax for the IMAGE\_BASE statement is as follows:

**IMAGE\_BASE**

### **The DEBUG Statement**

Specifies the TopSpeed debug information is included. This statement is not generated from the Clarion environment, but may be specified by manually editing the .EXP file and running the linker standalone. The syntax for the DEBUG statement is as follows:

**DEBUG**

### **The LINENUMBERS Statement**

Specifies that line number information in CodeView format is included. This statement is not generated from the Clarion environment, but may be specified by manually editing the .EXP file and running the linker standalone. The syntax for the IMAGE\_BASE statement is as follows:

**LINENUMBERS**

### **The SECTION\_ALIGNMENT Statement**

Specifies the section alignment must be in multiples of 4096. This statement is not generated from the Clarion environment, but may be specified by manually editing the .EXP file and running the linker standalone. The syntax is as follows:

**SECTION\_ALIGNMENT**

## The FILE\_ALIGNMENT Statement

Specifies the file alignment must be in multiples of 512. This statement is not generated from the Clarion environment, but may be specified by manually editing the .EXP file and running the linker standalone. The syntax is as follows:

**FILE\_ALIGNMENT**

## The EXPORTS Statement

The EXPORTS statement defines the names and attributes of the functions exported to other modules, and of the functions that run with I/O privilege. The term “export” refers to the process of making a function available to other run-time modules. By default, functions are hidden from other modules at run time.

**EXPORTS**  
**exportdefinitions**

The EXPORTS keyword marks the beginning of the export definitions. It may be followed by up to 3072 export definitions, each on a separate line. You should give an export definition for each dynamic-link routine that you want to make available to other modules. The syntax for an export definition is as follows:

**entryname [pwords] @number | ? [NODATA]**

|                  |                                                                                                                                                                                                                                                                                                                                                                                   |
|------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>entryname</i> | Defines the function name as it is known to other modules.                                                                                                                                                                                                                                                                                                                        |
| <i>pwords</i>    | Specifies the total size of the function's parameters, as measured in words (the total number of bytes divided by two). This field is required only if the function executes with I/O privilege. When a function with I/O privilege is called, OS/2 consults the pwords field to determine how many words to copy from the caller's stack to the I/O-privileged function's stack. |
| @number   ?      | Defines the function's ordinal position within the module-definition table. The @ may be followed by the position number of the function, or it may be followed by a question mark ( ? ) if the position is unknown. The numbers must be in sequence.                                                                                                                             |
| NODATA           | Provided for use by real-mode Windows (optional).                                                                                                                                                                                                                                                                                                                                 |

The EXPORTS statement is meaningful for functions within dynamic link libraries, functions which execute with I/O privilege, and call back functions in Windows programs.

For example:

```
EXPORTS
 Func1 @?
 Func2 @?
 CharTest @?
```

## Exporting CLASSES

Exporting CLASS declarations requires a special form of export definition.

You must create two export definitions for the CLASS itself. The first begins with VMT\$ followed by the name of the CLASS as the *entryname*. The second begins with TYPE\$ followed by the name of the CLASS as the *entryname*. These are followed by an export definition for each method in the CLASS to export whose *pwords* must begin with the name of the CLASS as the first parameter.

For example:

```
EXPORTS
 VMT$MYCLASS @?
 TYPE$MYCLASS @?
 FIRSTMETHOD@F7MYCLASS @?
 FIRSTMETHOD@F7MYCLASS @?
```

# 13 - MULTI LANGUAGE PROGRAMMING

## Overview

TopSpeed has both 16-bit and 32-bit (when released) C++ and Modula-2 compilers that can integrate into the Clarion environment. Additionally, 16-bit C and Pascal compilers are also available. The 16-bit compilers generate object code targeted at traditional Windows installations (3.1 and 3.11), while the 32-bit compilers generate object code for the Windows 95 and Windows NT environments. You must be running your development environment under a 32-bit environment to generate applications for a 32-bit environment.

Clarion also has object code generation capability that rivals that of many C compilers. You can then enhance that application with any low-level functions you need. These 3rd Generation Language (3GL) compilers enable the developer to include 3GL code modules directly into a Clarion project, giving unparalleled functionality and versatility. This mix of a Rapid Application Development (RAD) 4GL (4th Generation Language)—Clarion—and traditional 3GL compilers, makes Clarion an exceptional application development tool.

So why use C, C++, Pascal, or Modula-2 code at all in a Clarion application? Because there are libraries available (statistical, financial, graphics, communications, and many more) which could significantly cut the development time of an application which requires these capabilities. Many of these libraries are written in C, and many powerful C++, Pascal, and Modula-2, libraries are also available. Clarion allows you to use these libraries in their “native” form without “re-inventing the wheel.”

**Throughout this essay, we assume you have a good knowledge of the Clarion development environment, the Clarion language, and the 3GL in question.** The code examples assume that you use a TopSpeed compiler (other compiler’s requirements are also discussed). Since Clarion uses TopSpeed code generation and linking technology, it is easiest to link code produced with TopSpeed Compilers to Clarion applications. Clarion can link code produced by other third party compilers, however, some care is required as well as a good understanding of the operation of both compilers. It is not generally possible to directly link C++ code produced by C++ compilers other than TopSpeed (the reasons are explained later in this document).

## Compiler Integration

With the TopSpeed 3GL compilers installed, the Clarion development environment takes all the action necessary to call the correct compiler for each source module in the application. You cannot mix languages in a single source module, however, an application can contain any number of source modules written in any of the 3GLs or Clarion.

The development environment calls the correct compiler for each module at compile time by looking at the source file extensions, as follows:

| <u>Source File Extension</u> | <u>Compiler Called</u>   |
|------------------------------|--------------------------|
| .CLW                         | Clarion (16 and 32-bit)  |
| .CPP or .C                   | C++ (32-bit)             |
| .CPP                         | C++ (16-bit)             |
| .C                           | C (16-bit)               |
| .MOD                         | Modula-2 (16 and 32-bit) |
| .PAS                         | Pascal (16-bit only)     |

Source files with any other extensions will generate an 'Unknown compiler for ...' error message at compile time.

The Development Environment will also ensure that all modules are linked correctly and that the TopSpeed SmartLinker is given all the information that it requires.

## Integrating 3GL Modules into Clarion Projects

---

### Using the Application Generator

1. Create an "Include file" containing the function prototypes for the Clarion compiler.

You **MUST** prototype your functions if you intend to call them from Clarion code (see *Procedure Prototyping* in the *Language Reference*). The include file should contain prototypes for all 3GL functions called from Clarion code. Each 3GL module should have its own include file.

The include file is put in the Generated source without modification by step 3 of this process. The generated Clarion code for your include file appears in the Global Map something like this:

```
MODULE('module name')
 INCLUDE(' YourInc.Inc ')
END
```

The Application Generator generates the MODULE and END statements. Failure to correctly prototype your functions will almost certainly result in a General Protection Fault at run time.

2. Use the Text Editor to write your 3GL code.

Be sure to save the code with a file extension that the compiler can recognize (.C, .CPP, .MOD, or .PAS).

3. Add the module to the application as an **External Source Module**.

Select **Application ► Insert Module** from the main menu. Then select class **External Source** from the **Select Module Type** dialog. Enter the name of the 3GL module in the **Name** field, then enter the name of the include file in the **Map Include File** field of the **Module Properties** dialog

4. Compile and run the application.

### In a hand-coded Clarion application:

1. Create the function prototypes for the Clarion compiler.

You must prototype the functions you intend to call from Clarion code (see *Procedure Prototyping* in the *Language Reference*). The global MAP structure should contain the prototypes.

Each 3GL module should have its prototypes in a separate MODULE structure, something like this:

```
MODULE('module name')
 MyFunc(*CSTRING),CSTRING,RAW,PASCAL,NAME('_MyFunc')
END
```

You must include a complete MODULE structure in your Clarion MAP for all your 3GL modules. Failure to correctly prototype your functions will almost certainly result in a General Protection Fault at run time.

2. Use the Text Editor to write your 3GL code, saving the code with a file extension that the compiler can recognize (.C, .CPP, .MOD, or .PAS).

3. Add the module to the Project as an **External Source File**.

Select **Project ► Edit** from the main menu. Highlight **External Source Files** then press the **Add File...** button. Select the 3GL source module from the standard file open dialog that appears.

4. Compile and run the application.

## Resolving Data Types

The Clarion language defines the data types BYTE, SHORT, USHORT, LONG, ULONG, SREAL, REAL, and STRING which map fairly easily to C, C++, Pascal, and Modula-2 equivalents. Clarion also defines DATE and TIME data types, and GROUP structures, which may be mapped to structures in each language. CSTRING and PSTRING data types are specifically provided by Clarion to simplify interfacing with external functions using C or Pascal conventions.

The DECIMAL, PDECIMAL, BFLOAT4, and BFLOAT8 types are not discussed because it is very unlikely that these types of variables will ever be used in C, C++, Pascal, or Modula-2 code. If data of any of these types does need to be passed to C, C++, Pascal, or Modula-2 code, simply assign the value to a REAL or SREAL variable and pass that to the function (data type conversion is automatically handled in Clarion by the assignment statement).

The table below gives a brief cross reference of the parameters types supported by the Clarion, C++ and Modula-2 compilers; as detailed, some parameters require additional pragma statements to work correctly. The Clarion SIGNED and UNSIGNED data types are equates that change type from SHORT and USHORT to LONG and ULONG depending upon 16-bit or 32-bit compilation.

| <u>Clarion</u>         | <u>C++</u>                 | <u>Modula-2</u>            |
|------------------------|----------------------------|----------------------------|
| BYTE                   | unsigned char              | BOOLEAN                    |
| BYTE                   | unsigned char              | SHORTCARD                  |
| *BYTE                  | unsigned char *            | var SHORTCARD              |
| USHORT                 | unsigned short             | CARDINAL                   |
| *USHORT                | unsigned short *           | var CARDINAL               |
| SHORT                  | short                      | INTEGER                    |
| *SHORT                 | short *                    | var INTEGER                |
| LONG                   | long                       | LONGINT                    |
| *LONG                  | long *                     | var LONGINT                |
| ULONG                  | unsigned long              | LONGCARD                   |
| *ULONG                 | unsigned long *            | var LONGCARD               |
| SREAL                  | float                      | REAL                       |
| *SREAL                 | float *                    | var REAL                   |
| REAL                   | double                     | LONGREAL                   |
| *REAL                  | double *                   | var LONGREAL               |
| STRING                 | <i>can't pass by value</i> | <i>can't pass by value</i> |
| *STRING                | unsigned int, char *       | CARDINAL, ARRAY OF CHAR    |
| *STRING<br>(with RAW)  | char[]                     | var ARRAY OF CHAR          |
| *CSTRING<br>(with RAW) | char[] or char *           | var ARRAY OF CHAR          |
| *PSTRING               | char[] or Char *           | ARRAY OF CHAR              |
| GROUP                  | struct                     | var record type            |
| *GROUP<br>(with RAW)   | struct *                   | var record type            |
| *?                     | void far*                  | FarADDRESS                 |
| UNSIGNED               | unsigned int               |                            |
| SIGNED                 | int                        |                            |

Clarion `STRING` variables are normally passed as two parameters: first, a `UNSIGNED` which contains the length of the data buffer; second, the address of the data. `CSTRINGs` and `PSTRINGs` are passed the same as `STRINGs` (as two parameters). The `RAW` attribute can be used in the Clarion prototype to pass only the address of the string data to external 3GL functions (Clarion language procedures do not need, or support, `RAW`).

## C and C++ Data Type Equivalents

---

The following data type equivalents can be used with C or C++ code. These typedefs should appear in the `.H` header file referenced by the C or C++ code. The `CLA` prefix is used to avoid name clashes with third party libraries.

```
typedef unsigned char CLABYTE;
typedef short CLASHORT;
typedef unsigned short CLAUSHORT;
typedef long CLALONG;
typedef unsigned long CLAULONG;
typedef float CLASREAL;
typedef double CLAREAL;
```

Clarion `DATE` and `TIME` data types may be passed to C functions as a `CLALONG`, the `CLADATE` and `CLATIME` unions can then be used to resolve the elements of the date or time from the `CLALONG` value.

```
typedef union {
 CLALONG n;
 struct {
 CLABYTE ucDay;
 CLABYTE ucMonth;
 CLAUSHORT usYear;
 } s;
} CLADATE;
typedef union {
 CLALONG n;
 struct {
 CLABYTE ucHund;
 CLABYTE ucSecond;
 CLABYTE ucMinute;
 CLABYTE ucHour;
 } s;
} CLATIME;
```

Because of Clarion's two parameter method of passing `STRINGs`, the `CLASTRING` structure is useful for certain internal uses, but cannot be used to accept parameters from Clarion:

```
typedef struct {
 char *pucString;
 CLAUSHORT usLen;
} CLASTRING;
```

Clarion `STRING` variables are not `NULL` terminated, they are padded with spaces up to the length of the data buffer. The trailing spaces can be removed by using the Clarion `CLIP` procedure. The following code declares a `STRING` of 20 characters, assigns some data into it, and passes it as a parameter to a C or C++ function.

```
StringVar STRING(20)
CODE
StringVar = 'Hello World...'
C_Write_Function(StringVar)
```

The C or C++ function might be defined as:

```
extern void C_Write_Function(CLAUSHORT usLen, char *bData)
{ CLAUSHORT usNdx = 0;

 while (usNdx < usLen)
#ifdef __cplusplus
 cout << bData[usNdx++];
#else
 putchar(bData[usNdx++]);
#endif
}
```

In the above example, `usLen` would have a value of 20 and `bData` would be padded with trailing spaces. This padding would be written to the screen by `C_Write_Function()`. Many C routines expect a string to be `NULL` terminated. To address this issue, Clarion provides the `CSTRING` data type. `CSTRING` variables are automatically `NULL` terminated when data is assigned to them. This makes it possible for existing C routines to operate on the data.

A Clarion `GROUP` may be declared to contain related data. A group is roughly equivalent to a C or C++ struct. When passed as a parameter to a procedure, `GROUPS` are normally passed as three parameters: first, an `UNSIGNED` is passed which contains the size of the `GROUP`; second, the address of the `GROUP` structure; and third, the address of a buffer containing a type descriptor for the `GROUP`. The contents of the type descriptor are not discussed here and are subject to change in future versions of Clarion. `GROUPS` may be nested, and other `GROUPS` may be defined to assume the same structure as a previously declared `GROUP`. There are several forms of declaration for Clarion `GROUPS`:

```
Struct1 GROUP ! Struct1 is defined as a GROUP
u11 ULONG ! containing two ULONG values
u12 ULONG
END
```

This form of definition reserves space for `Struct1` and is equivalent to the C definition:

```
struct {
 CLAULONG u11;
 CLAULONG u12;
} Struct1;
```

In the following example, the declaration of Struct2 declares a GROUP similar to that defined by Struct1, however no space is reserved. In practice there need not be any instances of Struct2 defined.

```
Struct2 GROUP,TYPE ! Struct2 is declared as a GROUP
u13 ULONG ! containing two ULONG values
u14 ULONG
 END
```

The corresponding C definition is:

```
typedef struct {
 CLAULONG u13;
 CLAULONG u14;
} Struct2
```

In the following example, the definitions of Struct3 and Struct4 define them to be LIKE(Struct2), i.e. of the same internal structure. In order to distinguish members of Struct3 and Struct4 from those of Struct2 the S3 and S4 prefixes must be used. Struct3 and Struct4 define instances of Struct2 (which is not necessarily defined anywhere). In both cases space is reserved.

```
Struct3 LIKE(Struct2)
Struct4 LIKE(Struct2)
```

The corresponding C definitions are:

```
typedef Struct2 Struct3;
typedef Struct2 Struct4;

Struct3 S3;
Struct4 S4;
```

Clarion GROUP declarations may be nested, for example:

```
Struct5 GROUP,TYPE ! Struct5 is defined as a GROUP
Struct6 GROUP ! containing a nested GROUP
u15 ULONG
u16 ULONG
 END
 END
```

The equivalent C declaration is:

```
typedef struct {
 struct {
 CLAULONG u15;
 CLAULONG u16;
 } Struct6;
} Struct5;
```

## Modula-2 Data Type Equivalents

The following data type equivalents are used with Modula-2 code. These definitions should appear in the Modula-2 definition module referenced by the Modula-2 code. These should be used to define parameter and return types of procedures that will be called from Clarion code.

```

CONST
 BYTE ::= BYTE;
 SHORT ::= INTEGER (16-bit);
 USHORT ::= CARDINAL (16-bit);
 LONG ::= LONGINT;
 ULONG ::= LONGCARD;
 SREAL ::= REAL;
 REAL ::= LONGREAL;

```

Clarion DATE and TIME data types may be passed to Modula-2 procedures as a LONG, the DATE and TIME RECORDs can then be used to resolve the elements of the date or time from the LONG value.

```

DATE = RECORD
 CASE : BOOLEAN OF
 | TRUE:
 1 : LONG;
 ELSE
 ucDay : BYTE;
 ucMonth : BYTE;
 usYear : SHORT;
 END
 END;
TIME = RECORD
 CASE : BOOLEAN OF
 | TRUE:
 1 : LONG;
 ELSE
 ucHund : BYTE;
 ucSecond : BYTE;
 ucMinute : BYTE;
 ucHour : BYTE;
 END
 END;

```

Clarion STRINGS are passed in the same manner as Modula-2 open ARRAY OF CHAR parameters with the call(o\_a\_copy=>off) pragma in effect (the length and the address of the string are passed).

The following example code declares a string of 20 characters, assigns some data into it and passes it as a parameter to a Modula-2 procedure

```

MAP
 MODULE('M2_Code')
 M2_Write_Proc(*STRING), NAME('M2_Code$M2_Write_Proc')
 END
END
StringVar STRING(20)
CODE
StringVar = 'Hello World...'
M2_Write_Proc(StringVar)

```

The Modula-2 procedure might be defined as:

```
DEFINITION MODULE M2_Code;
(*# save, call(o_a_copy=>off) *)
PROCEDURE M2_Write_Proc(StringVar: ARRAY OF CHAR);
(*# restore *)
END M2_Code.
```

Note that Clarion STRINGS are not NULL terminated, they are padded with spaces up to the length of the data buffer. In the above example, StringVar would be padded with spaces up to a length of 20 characters. Variables of type CSTRING are automatically NULL terminated when data is assigned to them. This makes it possible for existing Modula-2 routines to operate on the data.

A Clarion GROUP is roughly equivalent to a Modula-2 RECORD. There are several forms of declaration for Clarion GROUPS. The following conforms to the Modula-2 declaration of the DATE type above:

```
DateType GROUP
n LONG
d GROUP,OVER(n)
ucDay BYTE
ucMonth BYTE
usYear SHORT
 END
 END
```

The OVER attribute is used to ensure that n and d occupy the same memory, the total size of the group is the size of the member n. When passed as parameters, GROUPs are normally passed as three parameters: first, an UNSIGNED is passed which contains the size of the GROUP; second, the address of the GROUP structure, and third, the address of a buffer containing a type descriptor for the GROUP. The contents of the type descriptor are not discussed here and are subject to change in future versions of Clarion. You may use the RAW attribute in your Clarion prototype for the Modula-2 procedure to instruct the compiler to pass only the address of the GROUP, otherwise you must define your Modula-2 procedure to take 2 extra parameters:

```
MAP
MODULE('M2_Code')
M2_Proc1(*GROUP)
M2_Proc2(*GROUP), RAW
END
END
```

The corresponding Modula-2 definition module would contain:

```
DEFINITION MODULE M2_Code;
TYPE
GROUP = RECORD
(* Members *)
END;
PROCEDURE M2_Proc1(Len: USHORT; VAR Data: GROUP; TypeDesc: ADDRESS);
PROCEDURE M2_Proc2(VAR Data: GROUP);
END M2_Code.
```

## Pascal Data Type Equivalents

The following data type equivalents can be used with Pascal code. These should be placed in the Pascal interface unit referenced by the Pascal code. These should be used to define parameter and return types of procedures that will be called from Clarion code.

```

ALIAS
SHORT = INT16;
USHORT = INT16;
LONG = INTEGER;
ULONG = INTEGER;
SREAL = SHORTREAL;

```

Clarion DATE and TIME data types may be passed to Pascal procedures as a LONG, the DATE and TIME records can then be used to resolve the elements of the date or time from the LONG value.

```

DATE = RECORD
CASE BOOLEAN OF
TRUE:
 (n : LONG);
FALSE:
 (ucDay : BYTE;
 ucMonth : BYTE;
 usYear : SHORT);
END;

```

```

TIME = RECORD
CASE BOOLEAN OF
TRUE:
 (n : LONG);
FALSE:
 (ucHund : BYTE;
 ucSecond : BYTE;
 ucMinute : BYTE;
 ucHour : BYTE);
END;

```

Because of Clarion's two parameter method of passing STRINGS, the STRING structure is useful for certain internal uses, but cannot be used to accept parameters from Clarion:

```

TYPE
STRING = RECORD
 usLen : USHORT;
 pucString : ^CHAR;
END;

```

Clarion PSTRINGS are passed by address in the same manner as Pascal STRING parameters with the call(s\_copy=>off) pragma in effect (the length and the address of the string are passed).

The following example code declares a string of 20 characters, assigns some data into it and passes it as a parameter to a Pascal procedure:

```

MAP
 MODULE('Pas_Code')
 Pas_Write_Proc(*PSTRING), NAME('Pas_Code$Pas_Write_Proc')
 END
END
StringVar PSTRING(20)
CODE
StringVar = 'Hello World...'
Pas_Write_Proc(StringVar)

```

The Pascal procedure might be defined as:

```

INTERFACE UNIT Pas_Code;

(*# save, call(s_copy=>off) *)
PROCEDURE Pas_Write_Proc(StringVar: STRING[HIGH]);
(*# restore *)
END.

```

A Clarion **GROUP** is roughly equivalent to a Pascal **RECORD**. There are several forms of declaration for Clarion **GROUPS**. The following duplicates the Pascal declaration of the **DATE** type above:

```

DateType GROUP
n LONG
d GROUP,OVER(n)
ucDay BYTE
ucMonth BYTE
usYear SHORT
 END
 END

```

The **OVER** attribute is used to ensure that **n** and **d** occupy the same memory, the total size of the group is the size of the member **n**. When passed as parameters, **GROUPS** are normally passed as three parameters: first, a **USHORT** is passed which contains the size of the **GROUP**; second, the address of the **GROUP** structure; and third, the address of a buffer containing a type descriptor for the **GROUP**. The contents of the type descriptor are not discussed here and are subject to change in future versions of Clarion. You may use the **RAW** attribute in your Clarion prototype for the Pascal procedure to instruct the compiler to pass only the address of the **GROUP**, otherwise you must define your Pascal procedure to take 2 extra parameters:

```

MAP
 MODULE('Pas_Code')
 Pas_Proc1(*GROUP)
 Pas_Proc2(*GROUP), RAW
 END
END

```

The corresponding Pascal interface unit might be:

```

INTERFACE UNIT Pas_Code;
TYPE
 GROUP = RECORD
 (* Members *)
 END;
PROCEDURE Pas_Proc1(Len: USHORT; VAR Data: GROUP; VAR TypeDesc);
PROCEDURE Pas_Proc2(VAR Data: GROUP);
END.

```

## Prototyping 3GL Functions in Clarion

The only thing necessary to be able to use any of the C standard library functions in Clarion code is the addition of the function's Clarion language prototype to the Clarion application's MAP structure. The Clarion prototype tells the compiler and linker what type of parameters are passed and what return data type (if any) to expect from the C function. The *PROCEDURE Prototypes* section in Chapter 2 of Clarion's *Language Reference* discusses the syntax and attributes required to create a prototype of a Clarion procedure. This same syntax is used to create Clarion prototypes of C functions.

There are four major issues involved in creating a prototype for a C function: calling convention, naming convention, parameter passing, and return data types from functions.

The calling convention for all the TopSpeed C standard library functions is the same register-based calling convention used by Clarion. Therefore, there is no need to use the C or PASCAL attributes in any standard C library function's Clarion prototype.

The TopSpeed C compiler's naming convention is the normal C convention. This means an underscore is automatically prepended to the function name when compiled. The Clarion NAME attribute is usually used in the prototype to give the linker the correct reference to a C function without requiring the Clarion code to use the prepended underscore. For example, the C function "access" is actually named "\_access" by the compiler. Therefore, the NAME('\_access') attribute is required in the prototype (unless you want to refer to the function in Clarion code as "\_access").

Each parameter passed to a C function must appear in its Clarion prototype as the data type of the passed parameter. Parameters are passed in Clarion either "by value" or "by address."

When a parameter is passed "by value," a copy of the data is received by the function. The passed parameter is represented in the prototype as the data type of the parameter. When passed "by address," the memory address of the data is received by the function. The parameter is represented in the prototype as the data type of the parameter with a prepended asterisk (\*). This corresponds to passing the C function the pointer to the data.

## Parameter Data Types

Parameter data type translation is the “key” to prototyping C functions in Clarion. The following is a table of C data types and the Clarion data type which should be used in the prototype:

| <u>C Data Type</u> | <u>Clarion Data Type</u>                  |
|--------------------|-------------------------------------------|
| char               | BYTE (gets linker warnings - ignore them) |
| unsigned char      | BYTE                                      |
| int                | SHORT                                     |
| unsigned int       | USHORT                                    |
| short              | SHORT                                     |
| unsigned short     | USHORT                                    |
| long               | LONG                                      |
| unsigned long      | ULONG                                     |
| float              | SREAL                                     |
| double             | REAL                                      |
| unsigned char *    | *BYTE                                     |
| int *              | *SHORT                                    |
| unsigned int *     | *USHORT                                   |
| short *            | *SHORT                                    |
| unsigned short *   | *USHORT                                   |
| long *             | *LONG                                     |
| unsigned long *    | *ULONG                                    |
| float *            | *SREAL                                    |
| double *           | *REAL                                     |
| char *             | *CSTRING w/ RAW attribute                 |
| struct *           | *GROUP w/ RAW attribute                   |

Since the Clarion language does not have a signed BYTE data type, linker warnings (‘type inconsistency’) will result when you prototype a function which receives a char parameter. As long as you are aware that the C function is expecting a signed value, and correctly adjust the BYTE field’s bitmap to pass a value in the range -128 to 127, this warning may be safely ignored.

The RAW attribute must be used when a C function expects to receive the address of a CSTRING or GROUP parameter. By default, Clarion STRING, CSTRING, PSTRING, and GROUP parameters are passed (internally) to other Clarion procedures as both the address and length of the string. C functions do not usually want or need the length, and expect to receive only the address of the data. Therefore, the RAW attribute overrides this default.

If the C function returns void, there is no data returned and the function fits the definition of a Clarion PROCEDURE. If the C function does return data, it is prototyped with the actual data type returned and the function fits the definition of a Clarion PROCEDURE which returns a value and may be called as part of a condition, assignment, or parameter list.

## Return Data Types

---

Return data types from C functions are almost the same as passed parameters:

| <u>C Return Type</u> | <u>Clarion Return Type</u>                   |
|----------------------|----------------------------------------------|
| char                 | BYTE (gets linker warnings - ignore them)    |
| unsigned char        | BYTE                                         |
| int                  | SHORT                                        |
| unsigned int         | USHORT                                       |
| short                | SHORT                                        |
| unsigned short       | USHORT                                       |
| long                 | LONG                                         |
| unsigned long        | ULONG                                        |
| float                | SREAL                                        |
| double               | REAL                                         |
| unsigned char *      | *BYTE                                        |
| int *                | *SHORT                                       |
| unsigned int *       | *USHORT                                      |
| short *              | *SHORT                                       |
| unsigned short *     | *USHORT                                      |
| long *               | *LONG                                        |
| unsigned long *      | *ULONG                                       |
| float *              | *SREAL                                       |
| double *             | *REAL                                        |
| char *               | CSTRING (pointer automatically dereferenced) |
| struct *             | ULONG (gets linker warnings - ignore them)   |

As you can see, the Clarion return type for a char \* is CSTRING (not \*CSTRING as you might expect). This is because the Clarion compiler automatically dereferences the pointer to the data when the function returns (as it does with all the pointer return types).

Notice that the Clarion return data type for struct \* is ULONG. This will generate a “type inconsistency” linker warning. This occurs because the Clarion language does not use pointers, and the ULONG is a four-byte integer which can serve as a replacement for a pointer return type. The warning is not a problem and can be safely ignored. You would probably use memcpy() to get at the returned data.

## Passing Parameters

Clarion offers two distinct methods of passing parameters to functions or procedures: “passed by value” and “passed by address.”

“Passed by value” means that the calling code passes a copy of the data to the called function or procedure. The called code can then operate on the data without affecting the caller’s copy of the data. These parameters are specified by the parameter’s data type in the prototype.

“Passed by address” means that the calling code passes the address of the data to the called function or procedure. With this method, the called function or procedure can modify the caller’s data. These parameters are specified by prefixing the parameter’s data type with an asterisk (\*) in the prototype:

```
MAP
MODULE('My_C_Lib')
 Var_Parameter(*USHORT) ! Parameter passed by address
 Val_Parameter(USHORT) ! Parameter passed by value
END
END
```

These declarations represent the Clarion interface to the functions contained in the C library `My_C_Lib`. The following example are the equivalent C declarations:

```
void Var_Parameter(CLAUSHORT *uspVal);
void Val_Parameter(CLAUSHORT usVal);
```

Clarion parameters “passed by address” are equivalent to pointers to the relevant C type. Clarion “passed by value” parameters are passed in the same way as C and C++ value parameters.

The corresponding Modula-2 definition module would be:

```
DEFINITION MODULE M2_Code;

 IMPORT C1a;

 PROCEDURE Var_Parameter(VAR us: C1a.USHORT);
 PROCEDURE Val_Parameter(us: C1a.USHORT);
END M2_Code.
```

The corresponding Pascal interface unit would be:

```
INTERFACE UNIT Pas_Code;

 IMPORT C1a;

 PROCEDURE Var_Parameter(VAR us: C1a.USHORT);
 PROCEDURE Val_Parameter(us: C1a.USHORT);
END.
```

You cannot pass a Clarion `STRING` or `GROUP` by value. For this reason, you must pass `STRINGS` or `GROUPS` by address.

## Resolving Calling Conventions

---

Clarion uses the TopSpeed object code generator, so it uses the same efficient register-based parameter passing mechanism employed by all TopSpeed languages. If differing calling conventions are used by code compiled by third-party compilers, the results may be unpredictable. Typically, the application will fail at run-time.

To use code produced by compilers other than TopSpeed, you must ensure that either:

- 1) The other compiler generates code using Clarion's (TopSpeed's) parameter passing method, or,
- 2) That Clarion generates code using the other compiler's parameter method.

You must also ensure that none of the functions return floating-point data types. There is no standard of compatibility between compilers regarding this issue. For example, Microsoft C returns floating-point values in a global variable while Borland C returns them on the stack (TopSpeed also returns them on the stack but there is no guarantee of compatibility). Therefore, any functions from non-TopSpeed compilers which must reference floating point values and modify them should receive them "passed by address" and directly modify the value — do not have the function return the value.

Most other compilers don't provide Clarion-compatible parameter passing conventions, but do provide standard C and Pascal parameter passing mechanisms (passed on the stack). Clarion has the C and PASCAL procedure prototype attributes to specify stack-based parameter passing.

Most non-TopSpeed C and C++ compilers use a calling convention where parameters are pushed onto the stack from right to left (as read from the parameter list). The Clarion C attribute specifies this convention. Many C and C++ compilers also offer a Pascal calling convention where parameters are pushed left to right from the parameter list. This convention is also used by most other languages on the PC. The Clarion PASCAL attribute generates calls using this convention.

In most cases, the C and PASCAL attributes are used in conjunction with the NAME attribute. This is because many compilers prepend an underscore to function names where the C convention is in use, and uppercase function names where the PASCAL convention is in use (Clarion uppercases procedure names also). For example:

```
MAP
 MODULE('My_C_Lib')
 StdC_Conv(UNSIGNED, ULONG), C, NAME('_StdC_Conv')
 StdPascal_Conv(UNSIGNED, ULONG), PASCAL, NAME('STDPASCAL_CONV')
 END
END
```

When the `StdC_Conv` procedure is called, the `ULONG` parameter is pushed on the stack followed by the `UNSIGNED` parameter. When `StdPascal_Conv` is called, the `UNSIGNED` parameter is pushed followed by the `ULONG` parameter. You should be very careful that calling conventions match, otherwise the program may behave unpredictably. When interfacing with code produced by TopSpeed compilers, the C and PASCAL calling convention attributes are not necessary because Clarion uses the TopSpeed register-based calling conventions.

When writing TopSpeed C functions to be called from a Clarion program, the `CLA_CONV` macro (discussed above) should be used to select the correct naming conventions. The best way of achieving this is to declare any interface functions in a separate header (.H) file and to apply the conventions to these declarations. C++ functions must be declared using “Pascal” external linkage (also discussed above). Modula-2 and Pascal naming conventions are best handled by using the `NAME` attribute on the prototype.

## Resolving Naming Conventions

---

When linking code produced from different programming tools, it is essential to ensure that the proper naming conventions are used. If differing naming conventions are used, the linker will not be able to resolve references to a name within code (produced by one compiler) and its definition (within code produced by another compiler). In this case, no .EXE will be generated.

Many C compilers (including TopSpeed) prepend an underscore to the name of each function or variable name. The Clarion `NAME` attribute simplifies interfacing with code produced by these compilers by explicitly telling the Clarion compiler the function or procedure name to generate for the linker. This allows you to explicitly code the Clarion prototype to follow the C convention. For example:

```
MAP
 MODULE('My_C_Lib')
 StdStr_Parm(STRING), NAME('_StdStr_Parm')
 END
END
```

When the Clarion compiler encounters the `StdStr_Parm()` procedure, it generates the name `_StdStr_Parm` in the object code. Although Clarion names are not case sensitive, the name generated using the `NAME` attribute will appear exactly as specified.

The following C language macro defines the Clarion naming conventions. This macro can be used when declaring C functions to interface with Clarion in order to force the C compiler to generate names following the Clarion naming convention (no prepended underscore and all upper case).

```
#define CLA_CONV name(prefix=>"", upper_case=>on)
```

C++ compilers encode the return and parameter types of a procedure into the name that appears in the object code in a process known as ‘name mangling’.

Therefore, C++ compiled functions which may be called from Clarion can be declared within a 'extern "Pascal" {...};' modifier, which is the equivalent to the C language CLA\_CONV macro (which does not affect the name mangling employed by the C++ compiler). For example:

```
extern "Pascal" void Clarion_Callable_Proc(void);
```

A more flexible form of the above, allowing for compilation by either a C or C++ compiler, is:

```
#ifdef __cplusplus
extern "Pascal" { /* Force Clarion conventions in C++ */
#else
#pragma save, CLA_CONV /* Force Clarion conventions in C */
#endif

void Clarion_Callable_Proc(void); /* C or C++ declaration */

#ifdef __cplusplus
} /* Restore C++ conventions */
#else
#pragma restore /* Restore C conventions */
#endif
```

This form of declaration usually appears in a header file to be included by any interface code. It ensures that the correct conventions are used when compiled with a C or C++ compiler and eliminates the need to use the NAME attribute on the Clarion language prototype of the procedure or function.

Clarion is a case-insensitive language and the compiler converts the names of all procedures to upper-case. Modula-2 and Pascal, however, are case sensitive and also prefix the name of all procedure names with the name of the module in the form: MyModule\$MyProcedure. The way to resolve these differences is to use Clarion's NAME attribute to specify the full name of the Modula-2 or Pascal procedure to the Clarion compiler:

```
MAP
MODULE('M2_Code')
 M2_Proc1(*GROUP), RAW, NAME('M2_Code$M2_Proc2')
END
MODULE('Pas_Code')
 Pas_Proc1(*GROUP), RAW, NAME('Pas_Code$Pas_Proc2')
END
END
```

The corresponding Modula-2 definition module might be:

```
DEFINITION MODULE M2_Code;

TYPE
 GROUP = RECORD
 (* Members *)
 END;
PROCEDURE M2_Proc1(VAR Data: GROUP);
END M2_Code.
```

The corresponding Pascal interface unit might be:

```

INTERFACE UNIT Pas_Code;
TYPE
 GROUP = RECORD
 (* Members *)
 END;
PROCEDURE Pas_Proc1(VAR Data: GROUP);
END.

```

The naming conventions used by Clarion for data differ from those used for PROCEDURES, and are more complex. Therefore, the NAME() attribute should be used to generate a Modula-2 or Pascal-compatible name for any Clarion data that needs to be accessed between languages. Modula-2 and Pascal data names are case sensitive and prefixed with the name of the module and a '@' in the form: MyModule@MyProc.

## The EXTERNAL and DLL Attributes

---

The EXTERNAL attribute is used to declare Clarion variables and functions that are defined in an external library. The DLL attribute declares that an EXTERNAL variable or functions is defined in a Dynamic Link Library (DLL). The DLL attribute is necessary for 32-bit projects, and ignored in 16-bit projects.

These attributes provide Clarion programs with a means of accessing public data in external libraries. The compiler will not reserve space for any variables declared as EXTERNAL. For example:

```

typedef struct {
 unsigned long u1;
 unsigned long u2;
} StructType;
#ifdef __cplusplus
extern "C" { /* Use C naming conventions, which will require use */
#endif /* of the NAME attribute in the Clarion prototype */
StructType Str1; /* Define Str1 */
StructType Str2; /* Define Str2 */
#ifdef __cplusplus
} /* Restore C++ conventions */
#endif

```

The following Clarion declarations are all that is necessary to make Str1 and Str2 available to Clarion programs.

```

StructType GROUP,TYPE ! Declare a user defined type
u1 ULONG
u2 ULONG
 END
 ! Declare Str1 and Str2 which are defined in the C module
Str1 LIKE(StructType),NAME('_Str1'),EXTERNAL
Str2 LIKE(StructType),NAME('_Str2'),EXTERNAL

```

The NAME attribute is used to allow the linker to use the C naming convention when referencing Str1 or Str2.

## Programming Considerations

### Using C++ Class Libraries

---

There are some limitations which apply to accessing C++ code and data from Clarion. C++ is an object oriented language and includes language features to support classes and objects, polymorphism, operator and function overloading, and class inheritance. None of these features are supported in Clarion as they are in C++. This does not prevent you from taking advantage of these features in a mixed Clarion and C++ application, but it does dictate the nature of the interface code.

Clarion cannot directly access C++ classes, or objects of a class type. Therefore, Clarion programs do not have direct access to the data or functions contained within those classes. To access them, it is necessary to provide a “C-like” interface to the C++ functionality. A C style function can be called from Clarion, which would then be able to access the C++ classes and objects defined within the code, including their public data and methods.

The following example code fragment demonstrate how to code a C++ function which calls a C++ class library. The MakeFileList function may be called directly from Clarion — the DirList constructors and the ReOrder class member may not. The DirList class implements a directory list whose entries may be ordered by name, size or date. The class definition and Clarion callable entry point declarations are as follows (note the use of the ‘extern “C”’ linkage specifier to force C naming conventions for the Clarion callable functions):

```
 /*** DirList Class Definition
class DirList: public List {
 public:
 DirList(char *Path, CLAUSHORT Attr, CLAUSHORT Order);
 DirList();
 void ReOrder(int Order);
};

 /*** Clarion Entrypoint Declarations
extern "C" {
 void MakeFileList(char *Path, CLAUSHORT Attr, CLAUSHORT Order);
}
```

The following code does nothing more than provide entry points for the Clarion code to access the functionality of the DIRLIST class library. Since Clarion performs no name-mangling and cannot access classes or their members, this API is necessarily fairly simple.

```

DirList *FileList = NULL; // The directory list object

void MakeFileList(char *Path, CLAUSHORT Attr, CLAUSHORT Order)
{ if (FileList != NULL) // If we have a list
 { delete FileList; // invoke class destructor
 FileList = NULL; // so we can start again
 }
 FileList = new DirList(Path, Attr, Order);
}

```

The following is the corresponding MAP structure prototype to allow Clarion to call the MakeFileList interface function:

```

MAP
MODULE('DirList')
 MakeFileList(*CSTRING,USHORT,USHORT),RAW,NAME('_MakeFileList')
END
END

```

One disadvantage of this is that, given a large class library, it appears to involve a lot of extra work to create a suitable interface. In practice, however, it should only be necessary to provide a very small interface to begin taking advantage of an existing C++ class library.

It is not possible to call C++ code compiled using non-TopSpeed C++ compilers from a Clarion application. C++ modules usually require special initialization — constructors for all static objects must be invoked in the correct order. This initialization process must be performed by the Clarion start-up code. Clarion's startup code automatically performs the necessary initialization for any TopSpeed C++ modules that are present, but it will not initialize modules compiled with other C++ compilers. Even if the modules did not require initialization, other C++ compilers use different calling and naming conventions, and adopt different internal class structures. This makes it impossible to use C++ class libraries in Clarion applications compiled with a compiler other than TopSpeed C++.

## Summary

---

The Clarion API provides a number of features to assist developers who need to interface to code written in other programming languages. With a little care, it is possible to create Clarion interfaces to some extremely powerful external libraries.

There is a complete example included with this document which includes code written in TopSpeed C, C++, Pascal, and Modula-2. This example code implements all the concepts discussed in this document. This is your resource which will “pull it all together” for you.

When preparing interfaces to libraries written in other languages you should consider the following suggestions:

- \* Don't write C, C++, Pascal, or Modula-2 functions to return CSTRING variables to Clarion. Have the other language routine place the CSTRING value in a public variable, or pass a \*CSTRING (by address) parameter to the C routine to receive the value.
- \* Don't call Clarion procedures which return STRING variables from other language functions. Have the Clarion procedure place the return value in a public variable or pass a \*CSTRING (by address) parameter to the other language procedure.
- \* For simplicity and efficiency, STRING and GROUP parameters should usually be passed by address with the RAW attribute to ensure only the address is passed.
- \* Test the application in XLARGE memory model first.

### **C and C++ Considerations**

- \* If a C or C++ function takes a pointer parameter, the corresponding parameter in the Clarion prototype for that function should be declared as "passed by address" by prefixing the data type with an asterisk (\*).
- \* If a C or C++ function takes a pointer to a GROUP, STRING, PSTRING or CSTRING, you should use the RAW attribute in the Clarion prototype.
- \* If a C or C++ function takes an ASCIIZ string as a parameter, the corresponding parameter in the Clarion prototype should be \*CSTRING.
- \* If a C or C++ function takes a pointer to a structure as a parameter, the corresponding parameter in the Clarion prototype should be \*GROUP.
- \* Use the header (.H) files as a template for developing a Clarion interface to a C or C++ library which eliminates the need to use the NAME attribute on the Clarion prototype to specify names.
- \* Use the NAME attribute on the Clarion prototype to specify names for C library functions which do not use the CLA\_CONV macro - remember that C names are case sensitive and start with an underscore (\_).

### **Modula-2 and Pascal Considerations**

- \* If a Modula-2 or Pascal procedure takes a VAR parameter, the corresponding parameter in the Clarion prototype for that procedure should be declared as "passed by address" by prefixing the data type with an asterisk (\*).
- \* If a Modula-2 or Pascal procedure takes a VAR parameter for a GROUP, STRING, PSTRING or CSTRING, you should use the RAW attribute in the Clarion prototype.
- \* If a Modula-2 or Pascal procedure takes a VAR record as a parameter, the corresponding parameter in the Clarion prototype should be \*GROUP and the RAW attribute should be used in the prototype.

### **Additional C++ Considerations**

- \* Use the “Pascal” external linkage specification for your C++ interface functions. This eliminates the need to use the Clarion NAME attribute on the prototype.
- \* Don’t call C++ class member functions from your Clarion code.
- \* Don’t try to access C++ objects of class type from your Clarion code.
- \* Don’t try to access C++ code compiled with a C++ compiler other than TopSpeed.

### **Additional Modula-2 Considerations**

- \* Use the definition (.DEF) module as a template for developing a Clarion interface to a Modula-2 library.
- \* If a Modula-2 procedure takes an ASCII string as a parameter, the corresponding parameter in the Clarion prototype should be \*CSTRING.
- \* Use the NAME attribute to specify names for Modula-2 library procedures -remember that Modula-2 names are prefixed with the module name followed by a ‘\$’ and are case-sensitive.

### **Additional Pascal Considerations**

- \* Use the interface (.ITF) files as a template for developing a Clarion interface to a Pascal library.
- \* Use the NAME attribute to specify names for Pascal library procedures - remember that Pascal names are prefixed with the module name followed by a ‘\$’ and are upper-case.



# 14 - API CALLS AND ADVANCED RESOURCES

## Prototypes and Declarations

On your installation CD, you have files with prototypes, declarations, and headers that you can use to let Clarion “talk” to Windows, C/C++, Modula-2, and vice versa.

### Clarion to C/C++ Standard Library

---

To call the standard C library functions from Clarion applications, include `\CLIB.CLW` in the “Inside the Global Map” embed point.

```
INCLUDE('CLIB.CLW')
```

This file contains Clarion prototypes for various string handling functions, integer math, character type functions, and low level file manipulation functions. Refer to your C/C++ Library Reference for more information on individual functions.

### Clarion to Windows API

---

To call Windows API functions from Clarion applications, you must include the functions’ prototypes in your application’s MAP structure, and any standard EQUATES or data structures that the functions need in your Global data declarations.

Clarion contains the WINAPI.EXE utility program that creates the file you need to include in your application. This program, by default, creates the WINAPI.CLW file which has two sections: the “Equates” section containing all EQUATE statements and any data structures needed by the functions you choose, and the “Prototypes” section containing the Clarion language prototypes of Windows API functions you choose to use.

Include the Equates section of WINAPI.CLW in the “After Global INCLUDEs” embed point:

```
INCLUDE('WINAPI.CLW', 'Equates')
```

Include the Prototypes section of WINAPI.CLW in the “Inside the Global Map” embed point:

```
INCLUDE('WINAPI.CLW', 'Prototypes')
```

Refer to your Windows API reference for more information on the individual API functions available to you in categories such as:

Creating Windows

- Window Support
- Message Processing
- Memory Management
- Bitmaps and Icons
- Color Palette Control
- Sound
- Character Sets and Strings
- Communications
- Metafiles
- Tool Help Library
- File Compression
- Installation and Version Information
- TrueType Fonts
- Multimedia

## Modula-2 to Clarion

---

### Clarion's Runtime Library

To call the Clarion runtime library procedures, use the \CWRUN.DEF file. This file contains Modula-2 declarations for various Clarion Language procedures, as well as the many standard C library functions that are found in the CWRUNxx.DLL. The available functions are documented in the *Clarion's Runtime Library Functions* section of this article.

### Clarion's File Driver Procedures

To call the Clarion database file driver procedures, use the \CWFILE.DEF file. This file contains Modula-2 declarations for Clarion's FILE, RECORD, KEY, INDEX, MEMO, and BLOB handling procedures, including a complete description of Clarion's file control block.

## C/C++ to Clarion

---

### Clarion's Runtime Library

To call the Clarion runtime library procedures, use the \CWRUN.H file. This file contains C/C++ prototypes for various Clarion Language procedures, as well as many standard C library functions that are found in the CWRUNxx.DLL. The available functions are documented below in the *Clarion's Runtime Library Functions* section.

### Clarion's File Driver Procedures

To call the Clarion database file driver procedures, use the \CWFILE.H file. This file contains C/C++ prototypes for Clarion's FILE, RECORD, KEY, INDEX, MEMO, and BLOB handling procedures, including a complete description of Clarion's file control block.

## ***Accessing Clarion's Runtime Library from C/C++ or Modula-2 Code***

Following is a list of Clarion runtime library procedures, data structures, and variables, that you may use at run time in your C/C++ or Modula-2 code.

### **Structures and Data Type Definitions**

---

#### **COLORREF**

C++:           typedef unsigned long COLORREF;

Modula-2:      TYPE COLORREF = LONGINT;

### **Run-Time Variables**

---

The following variables are available for interrogation at run-time:

**Cla\$DOSerror**      An unsigned integer containing the last DOS error code.

**Cla\$FILEERRCODE**      An integer containing the last Clarion error code.

**Cla\$FILEERRORMSG**      A character array of 80 char's containing the last Clarion error message.

**WSL@AppInstance**      An unsigned short containing the instance ID of the application.

## Clarion Built-in Procedures

---

The following list of procedures are those internal Clarion procedures that are 'safe' to call at run-time. Unless otherwise stated, assume that these procedures have been given external C linkage.

|                  |                                                                                                                                                             |
|------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Cla\$ACOS</b> | The Clarion ACOS() procedure. Returns the inverse cosine of the val parameter.                                                                              |
| C++:             | double Cla\$ACOS(double val)                                                                                                                                |
| Modula-2:        | Cla\$ACOS(val :LONGREAL):LONGREAL;                                                                                                                          |
|                  | val:                   A numeric expression describing an angle in radians.                                                                                 |
| <b>Cla\$ARC</b>  | The Clarion ARC statement. Places an arc of an ellipse on the current window or report, bounded by the rectangle defined by the x, y, wd and ht parameters. |
| C++:             | void Cla\$ARC(int x, int y, int wd, int ht, int start, int end)                                                                                             |
| Modula-2:        | Cla\$ARC(x,y,wd,ht,start,end: INTEGER);                                                                                                                     |
|                  | x:                    An integer specifying the horizontal position of the starting point.                                                                  |
|                  | y:                    An integer specifying the vertical position of the starting point.                                                                    |
|                  | wd:                   An integer specifying then width.                                                                                                     |
|                  | ht:                   An integer specifying then height.                                                                                                    |
|                  | start:                An integer specifying the start of the arc in 10th's of a degree.                                                                     |
|                  | end:                   An integer specifying the end of the arc in 10th's of a degree.                                                                      |
| <b>Cla\$ASIN</b> | The Clarion ASIN() procedure. Returns the inverse sine of the val parameter.                                                                                |
| C++:             | double Cla\$ASIN(double val)                                                                                                                                |
| Modula-2:        | Cla\$ASIN(val LONGREAL): LONGREAL;                                                                                                                          |

|                    |            |                                                                                                                                                                                                        |
|--------------------|------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                    | val:       | A numeric expression describing an angle in radians.                                                                                                                                                   |
| <b>Cla\$ATAN</b>   |            | The Clarion ATAN() procedure. Returns the inverse tangent of the val parameter.                                                                                                                        |
| C++:               |            | double Cla\$ATAN(double val)                                                                                                                                                                           |
| Modula-2:          |            | Cla\$ATAN(val: LONGREAL):LONGREAL;                                                                                                                                                                     |
|                    | val:       | A numeric expression describing an angle in radians.                                                                                                                                                   |
| <b>Cla\$BOX</b>    |            | The Clarion BOX statement. This procedure draws a box of the color specified by the COLORREF structure, starting at position x, y of the width and height specified on the current window or report.   |
| C++:               |            | void Cla\$BOX(int x, int y, int wd, int ht, COLORREF fillcolor)                                                                                                                                        |
| Modula-2:          |            | Cla\$BOX(x, y, wd, ht: INTEGER; fillcolor: COLORREF);                                                                                                                                                  |
|                    | x:         | An integer specifying the horizontal start position.                                                                                                                                                   |
|                    | y:         | An integer specifying the vertical start position.                                                                                                                                                     |
|                    | wd:        | An integer specifying the width.                                                                                                                                                                       |
|                    | ht:        | An integer specifying the height.                                                                                                                                                                      |
|                    | fillcolor: | A COLORREF structure.                                                                                                                                                                                  |
| <b>Cla\$BSHIFT</b> |            | The Clarion BSHIFT() procedure. This procedure returns the result of bit shifting val by count binary positions. If count is positive, val is shifted left, if count is negative val is shifted right. |
| C++:               |            | long Cla\$BSHIFT(long val, int count)                                                                                                                                                                  |
| Modula-2:          |            | Cla\$BSHIFT(val: LONGINT; count: INTEGER): LONGINT;                                                                                                                                                    |
|                    | val:       | A numeric expression.                                                                                                                                                                                  |
|                    | count:     | A numeric expression.                                                                                                                                                                                  |

|                   |                                                                                                                                                                                                                                                                                    |
|-------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Cla\$CHORD</b> | The Clarion CHORD statement. Draws a closed sector ellipse on the current window or report inside the box specified by the x, y, wd and ht parameters and in the color provided in the COLORREF structure. The start and end parameters specify which part of the ellipse to draw. |
| C++:              | void Cla\$CHORD(int x, int y, int wd, int ht, int start, int end, COLORREF fillcolor)                                                                                                                                                                                              |
| Modula-2:         | Cla\$CHORD(x, y, wd, ht, start, end: INTEGER; fillcolor: COLORREF);                                                                                                                                                                                                                |
|                   | x: An integer specifying the horizontal start position.                                                                                                                                                                                                                            |
|                   | y: An integer specifying the vertical start position.                                                                                                                                                                                                                              |
|                   | wd: An integer specifying the width.                                                                                                                                                                                                                                               |
|                   | ht: An integer specifying the height.                                                                                                                                                                                                                                              |
|                   | start: An integer expressing the string of the chord in 10th's of a degree.                                                                                                                                                                                                        |
|                   | end: An integer expressing the end of the chord in 10th's of a degree.                                                                                                                                                                                                             |
|                   | fillcolor: A COLORREF structure.                                                                                                                                                                                                                                                   |
| <b>Cla\$CLOCK</b> | The Clarion CLOCK() procedure. Returns the system time in the form of a Clarion standard time.                                                                                                                                                                                     |
| C++:              | long Cla\$CLOCK(void)                                                                                                                                                                                                                                                              |
| Modula-2:         | Cla\$CLOCK(): LONGINT;                                                                                                                                                                                                                                                             |
| <b>Cla\$COS</b>   | The Clarion COS() procedure. Returns the cosine of the val parameter.                                                                                                                                                                                                              |
| C++:              | double Cla\$COS(double val)                                                                                                                                                                                                                                                        |
| Modula-2:         | Cla\$COS(val: LONGREAL): LONGREAL;                                                                                                                                                                                                                                                 |
|                   | val: A numeric expression describing an angle in radians.                                                                                                                                                                                                                          |

|                     |                                                                                                                                                                                               |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Cla\$DATE</b>    | The Clarion DATE() procedure. Returns a Clarion standard date value form the component day, month and year parameters.                                                                        |
| C++:                | long Cla\$DATE(unsigned mn, unsigned dy, unsigned yr)                                                                                                                                         |
| Modula-2:           | Cla\$DATE(mn, dy, yr: CARDINAL): LONGINT;                                                                                                                                                     |
|                     | mn: A numeric expression for the month in the range 1 to 12.                                                                                                                                  |
|                     | dy: A numeric expression for the day in the range 1 to 31.                                                                                                                                    |
|                     | yr: A numeric expression for the year in the range 1801 to 2099.                                                                                                                              |
| <b>Cla\$DAY</b>     | The Clarion DAY() procedure. Returns the day in the range 1 to 31 from the Clarion standard date parameter.                                                                                   |
| C++:                | long Cla\$DAY(long dt)                                                                                                                                                                        |
| Modula-2:           | Cla\$DAY(dt: LONGINT): LONGINT;                                                                                                                                                               |
|                     | dt: A numeric expression for Clarion standard date.                                                                                                                                           |
| <b>Cla\$ELLIPSE</b> | The Clarion ELLIPSE statement. Draws an ellipse on the current window or report, of the color specified in the COLORREF structure, inside the area bounded by the x, y, wd and ht parameters. |
| C++:                | void Cla\$ELLIPSE(int x, int y, int wd, int ht, COLORREF fillcolor)                                                                                                                           |
| Modula-2:           | Cla\$ELLIPSE(x, y, wd, ht: INTEGER; fillcolor: COLOREF);                                                                                                                                      |
|                     | x: An integer expression.                                                                                                                                                                     |
|                     | y: An integer expression.                                                                                                                                                                     |
|                     | wd: An integer expression.                                                                                                                                                                    |
|                     | ht: An integer expression.                                                                                                                                                                    |
|                     | fillcolor: A COLORREF structure.                                                                                                                                                              |

|                   |                                                                                                                                                          |
|-------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Cl\$INT</b>    | The Clarion INT() procedure. Returns the integer portion of the val parameter. The value is truncated at the decimal point and no rounding is performed. |
| C++:              | double Cl\$INT(double val)                                                                                                                               |
| Modula-2:         | Cl\$INT(val: LONGREAL): LONGREAL;                                                                                                                        |
|                   | val:           A numeric expression.                                                                                                                     |
| <b>Cl\$LOG10</b>  | The Clarion LOG10() procedure. Returns the base 10 logarithm of the val parameter.                                                                       |
| C++:              | double Cl\$LOG10(double val)                                                                                                                             |
| Modula-2:         | Cl\$LOG10(val: LONGREAL): LONGREAL;                                                                                                                      |
|                   | val:           A numeric expression.                                                                                                                     |
| <b>Cl\$LOGE</b>   | The Clarion LOGE() procedure. Returns the natural logarithm of the val parameter.                                                                        |
| C++:              | double Cl\$LOGE(double val)                                                                                                                              |
| Modula-2:         | Cl\$LOGE(val: LONGREAL): LONGREAL;                                                                                                                       |
|                   | val:           A numeric expression.                                                                                                                     |
| <b>Cl\$MONTH</b>  | The Clarion MONTH() procedure. Returns the month from a Clarion standard date in the range 1 to 12.                                                      |
| C++:              | long Cl\$MONTH(long dt)                                                                                                                                  |
| Modula-2:         | Cl\$MONTH(dt: LONGINT): LONGINT;                                                                                                                         |
|                   | dt:            A numeric expression containing a Clarion standard date.                                                                                  |
| <b>Cl\$MOUSEX</b> | The Clarion MOUSEX() procedure. Returns the horizontal position of the mouse.                                                                            |
| C++:              | int Cl\$MOUSEX(void)                                                                                                                                     |
| Modula-2:         | Cl\$MOUSEX(): INTEGER;                                                                                                                                   |

|                     |                                                                                                                                    |
|---------------------|------------------------------------------------------------------------------------------------------------------------------------|
| <b>Cla\$MOUSEY</b>  | The Clarion MOUSEY() procedure. Returns the horizontal position of the mouse.                                                      |
| C++:                | int Cla\$MOUSEY(void)                                                                                                              |
| Modula-2:           | Cla\$MOUSEY(): INTEGER;                                                                                                            |
| <b>Cla\$NUMERIC</b> | The Clarion NUMERIC() procedure. Returns 1 (true) if str contains a valid representation of a number, otherwise returns 0 (false). |
| C++:                | unsigned Cla\$NUMERIC(char *str, unsigned slen)                                                                                    |
| Modula-2:           | Cla\$NUMERIC(VAR str: ARRAY OF CHAR;<br>slen: CARDINAL): CARDINAL;                                                                 |
|                     | str:                   A pointer to a string.                                                                                      |
|                     | slen:                   Length of the str parameter.                                                                               |
| <b>Cla\$RANDOM</b>  | The Clarion RANDOM() procedure. Returns a pseudo-random number whose value will be between the low and high bound values.          |
| C++:                | long Cla\$RANDOM(long low, long high)                                                                                              |
| Modula-2:           | Cla\$RANDOM(low, high: LONGINT): LONGINT;                                                                                          |
|                     | low:                   A numeric value specifying the lower bound.                                                                 |
|                     | high:                   A numeric value specifying the upper bound.                                                                |
| <b>Cla\$ROUND</b>   | The Clarion ROUND() procedure. Returns the val parameter rounded to power of 10 specified by the ord parameter.                    |
| C++:                | double Cla\$ROUND(double val, double ord)                                                                                          |
| Modula-2:           | Cla\$ROUND(val, ord: LONGREAL):<br>LONGREAL;                                                                                       |
|                     | val:                   A numeric expression.                                                                                       |
|                     | ord:                   A numeric expression equal to a power of 10 (e.g. .001, .0, 1, 10, 100 etc...).                             |

- Cl\$SETCLOCK** The Clarion SETCLOCK statement. Sets the system clock to the time contained in the dt parameter.
- C++:           void Cl\$SETCLOCK(long dt)
- Modula-2:       Cl\$SETCLOCK(dt: LONGINT);
- dt:             A numeric expression representing a Clarion standard time.
- Cl\$SETTODAY** The Clarion SETTODAY statement. Sets the DOS system date to that contained in the dt parameter.
- C++:           void Cl\$SETTODAY(long dt)
- Modula-2:       Cl&SETTODAY(dt: LONGINT);
- dt:             A numeric expression containing a Clarion standard date.
- Cl\$SIN**        The Clarion SIN() procedure. Returns the sine of the val parameter.
- C++:           double Cl\$SIN(double val)
- Modula-2:       CL\$SIN(val: LONGREAL): LONGREAL;
- val:            A numeric expression describing an angle in radians.
- Cl\$SQRT**       The Clarion SQRT() procedure. Returns the square root of the val parameter.
- C++:           double Cl\$SQRT(double val)
- Modula-2:       Cl\$SQRT(val:LONGREAL): LONGREAL;
- val:            A numeric expression.
- Cl\$TAN**        The Clarion TAN() procedure. Returns the tangent of the val parameter.
- C++:           double Cl\$TAN(double val)
- Modula-2:       Cl\$TAN(val: LONGREAL): LONGREAL;
- val:            A numeric expression describing an angle in radians.

|                   |                                                                                                          |
|-------------------|----------------------------------------------------------------------------------------------------------|
| <b>Cla\$TODAY</b> | The Clarion TODAY() procedure. Returns the system date in Clarion standard date format.                  |
| C++:              | long Cla\$TODAY(void)                                                                                    |
| Modula-2:         | Cla\$TODAY(): LONGINT;                                                                                   |
| <b>Cla\$YEAR</b>  | The Clarion YEAR() procedure. Extracts the year from a Clarion standard date, in the range 1801 to 2099. |
| C++:              | long Cla\$YEAR(long dt)                                                                                  |
| Modula-2:         | Cla\$YEAR(dt: LONGINT): LONGINT;                                                                         |
|                   | dt: A numeric expression describing a Clarion standard date.                                             |

### Clarion String Stack Handling Procedures

The following section describes the use Clarion internal run-time string handling procedures available to 3GL code. Clarion uses a LISP like approach to string handling whereby, parameters are pushed onto the top of the string stack, with operations being performed on the topmost entries. Assume, unless otherwise documented, that the procedures remove (or Pop) items off the stack that they have used.

Please note that some of the following procedures require pointers to null terminated strings, to be passed as parameters. Modula-2 programmers should use the Modula library procedure Str.StrToC to convert strings to null terminated equivalents. Also, the pragma *call(o\_a\_size=>off,o\_a\_copy=>off)* must be issued to prevent the passing of array size information to the run-time procedures.

|                        |                                                                                                                                      |
|------------------------|--------------------------------------------------------------------------------------------------------------------------------------|
| <b>Cla\$PopCString</b> | Takes the topmost item off the stack and copies it to the string pointed to by s; len contains the length of the string copied to s. |
| C++:                   | void Cla\$PopCString(char *s, unsigned len)                                                                                          |
| Modula-2:              | Cla\$PopCString(s: POINTER TO CHAR; len: CARDINAL);                                                                                  |
|                        | s: A pointer to a null terminated string                                                                                             |
|                        | len: The length of string s                                                                                                          |

- Cla\$PopPString** Takes the topmost item off the stack and copies it to the string pointed to by *s*; *len* contains the length of the string copied to *s*. The string is converted to a Pascal style string (i.e. first byte is string length) during copy.
- C++:           void Cla\$PopPString(char \*s, unsigned len)
- Modula-2:      Cla\$PopPString(VAR s: ARRAY OF CHAR; len: CARDINAL);
- s*:                A pointer to a string
- len*:             The length of string *s*
- Cla\$PopString** Pops the uppermost stack item and copies it to the string *s*.
- C++:           void Cla\$PopString(char \*s, unsigned len)
- Modula-2:      Cla\$PopString(VAR s: ARRAY OF CHAR; len: CARDINAL);
- s*:                A pointer to a null terminated string
- len*:             The length of string *s*
- Cla\$PushCString** Pushes *s* onto the top of the stack.
- C++:           void Cla\$PushCString(char \*s)
- Modula-2:      Cla\$PushCString(VAR s: ARRAY OF CHAR);
- s*:                A pointer to a null terminated string
- Cla\$PushString** Pushes the string *s* onto the top of the stack. *Len* specifies the length of string *s*.
- C++:           void Cla\$PushString(char \*s, unsigned len)
- Modula-2:      Cla\$PushString(VAR s: ARRAY OF CHAR; len: CARDINAL);
- s*:                A pointer to a string
- len*:             The length of string *s*

**Cla\$StackALL** The Clarion ALL() procedure. Pops the top item of the stack and replaces it by a string containing the original string replicated as many times as necessary to produce a string of length len.

C++: void Cla\$StackALL(unsigned len)

Modula-2: Cla\$StackALL(len: CARDINAL);

len: An unsigned integer

### **Cla\$StackCENTER**

The Clarion CENTER() procedure. Pops the topmost item of the stack and replaces it with a string padded with leading spaces so as to center the text in a string of length len.

C++: void Cla\$StackCENTER(unsigned len)

Modula-2: Cla\$StackCENTER(len: CARDINAL);

len: An unsigned integer

**Cla\$StackCLIP** The Clarion CLIP() procedure. Removes trailing spaces from the top most item on the stack.

C++: void Cla\$StackCLIP(void)

Modula-2: Cla\$StackCLIP();

### **Cla\$StackCompare**

Compares the top item on the stack (s1) with the 2nd item on the stack (s2) and returns one of the following values:

-1: if  $s1 < s2$

0: if  $s1 = s2$

1: if  $s1 > s2$

After the compare instruction, s1 and s2 are removed from the stack automatically.

C++: int Cla\$StackCompare(void)

Modula-2: Cla\$StackCompare(): INTEGER;

**Cla\$StackCompareN**

Compares the topmost item on the stack to null. Returns true if the topmost item is null, otherwise returns false.

C++: int Cla\$StackCompareN(void)

Modula-2: Cla\$StackCompareN(): INTEGER;

**Cla\$StackConcat** Pops the top two items off the stack, concatenates them together and pushes the resulting string back onto the stack.

C++: void Cla\$StackConcat(void)

Modula-2: Cla\$StackConcat();

**Cla\$StackINSTRING**

The Clarion INSTRING() procedure. Searches the topmost item on the stack, for any occurrence of the second item on the stack. The search starts at character position start and increments the start position by step until the end of the string is reached. Returns the iteration count required to find the search string, or 0 if not found.

C++: unsigned Cla\$StackINSTRING(unsigned step, unsigned start)

Modula-2: Cla\$StackINSTRING(step, start: CARDINAL): CARDINAL;

step: An unsigned integer, the search increment

start: An unsigned integer, the start position of the search

**Cla\$StackLEFT** The Clarion LEFT() procedure. Replaces the topmost string on the stack with its left justified equivalent. The replacement string will have a length of len.

C++: void Cla\$StackLEFT(unsigned len)

Modula-2: Cla\$StackLEFT(len: CARDINAL);

len: An unsigned integer

|                         |                                                                                                                          |
|-------------------------|--------------------------------------------------------------------------------------------------------------------------|
| <b>Cl\$StackLen</b>     | Returns the length of the topmost item on the stack. Does not pop the item off the stack.                                |
| C++:                    | unsigned Cl\$StackLen(void)                                                                                              |
| Modula-2:               | Cl\$StackLen(): CARDINAL;                                                                                                |
| <b>Cl\$StackLen2</b>    | Returns the length of the topmost item on the stack. Pops the item off the stack after getting its length.               |
| C++:                    | unsigned Cl\$StackLen2(void)                                                                                             |
| Modula-2:               | Cl\$StackLen2(): CARDINAL;                                                                                               |
| <b>Cl\$StackLOWER</b>   | The Clarion LOWER() procedure. Replaces the topmost string on the stack with its lower case equivalent.                  |
| C++:                    | void Cl\$StackLOWER(void)                                                                                                |
| Modula-2:               | Cl\$StackLOWER();                                                                                                        |
| <b>Cl\$STACKpop</b>     | Pops the top item off the stack.                                                                                         |
| C++:                    | void Cl\$STACKpop(void)                                                                                                  |
| Modula-2:               | Cl\$STACKpop();                                                                                                          |
| <b>Cl\$StackNUMERIC</b> | Returns true if the topmost string on stack contains a valid numeric representation, otherwise returns false.            |
| C++:                    | unsigned Cl\$StackNUMERIC(void)                                                                                          |
| Modula-2:               | Cl\$StackNUMERIC(): CARDINAL;                                                                                            |
| <b>Cl\$StackPRESS</b>   | The Clarion PRESS statement. Pushes every character in the topmost string of the stack into the Windows keyboard buffer. |
| C++:                    | void Cl\$StackPRESS(void)                                                                                                |
| Modula-2:               | Cl\$StackPRESS();                                                                                                        |

**Cla\$StackRIGHT** The Clarion RIGHT() procedure. Replaces the topmost item on the stack with its right justified equivalent. The replacement string will have a length of len characters.

C++: void Cla\$StackRIGHT(unsigned len)

Modula-2: Cla\$StackRIGHT(len: CARDINAL);

len: An unsigned integer

**Cla\$StackSUB** The Clarion SUB() procedure. Replaces the topmost string on the stack with a sub slice of the string starting at character position pos and of length len.

C++: void Cla\$StackSUB(unsigned pos, unsigned len)

Modula-2: Cla\$StackSUB(pos, len: CARDINAL);

pos: An unsigned integer; the start position of the sub string

len: An unsigned integer; the length of the sub string

**Cla\$StackVAL** The Clarion VAL() procedure. Returns the ANSI value of the first character of the topmost string of the stack.

C++: unsigned char Cla\$StackVAL(void)

Modula-2: Cla\$StackVAL(): BYTE;

**Cla\$StackUPPER** Replace the topmost string on the stack with its uppercase equivalent.

C++: void Cla\$StackUPPER(void)

Modula-2: Cla\$StackUPPER();

## Standard C Functions in Clarion's Runtime Library

The following functions comprise a sub-set of the standard TopSpeed library that you can call from your Clarion, C/C++, or Modula-2 code. All of these functions are fully documented in the TopSpeed C Library Reference manual (or in any ANSI-standard C library reference) and so, are not documented here. Unless otherwise indicated, assume that the functions operate exactly as documented.

The purpose of this list is simply to let you know what C standard library functions are available and the correct prototypes for each language.

### Conversion Functions

---

Please note that some of the following functions require pointers to null terminated strings as parameters. Modula-2 programmers should use the Modula library procedure `Str.StrToC` to convert strings to null terminated equivalents. Also, the pragma `call(o_a_size=>off; o_a_copy=>off)` must be issued to prevent the passing of array size information to the run-time procedures.

|             |                                                         |
|-------------|---------------------------------------------------------|
| <b>atof</b> | Convert string to floating point.                       |
| C++:        | <code>double atof(const char *_nptr)</code>             |
| Modula-2:   | <code>atof( VAR _nptr: ARRAY OF CHAR): LONGREAL;</code> |
| Clarion:    | <code>AToF(*cstring),real,raw,name('_atof')</code>      |
| <b>atoi</b> | Convert string to integer.                              |
| C++:        | <code>int atoi(const char *_nptr)</code>                |
| Modula-2:   | <code>atoi(VAR _nptr: ARRAY OF CHAR): INTEGER;</code>   |
| Clarion:    | <code>AToI(*cstring),short,raw,name('_atoi')</code>     |
| <b>atol</b> | Convert string to long.                                 |
| C++:        | <code>long atol(const char *_nptr)</code>               |

|              |                                               |
|--------------|-----------------------------------------------|
| Modula-2:    | atol( VAR _nptr: ARRAY OF CHAR): LONGINT;     |
| Clarion:     | AToL(*cstring),long,raw,name('_atol')         |
| <b>atoul</b> | Convert string to unsigned long.              |
| C++:         | unsigned long atoul(const char *_nptr)        |
| Modula-2:    | atoul(VAR _nptr: ARRAY OF CHAR):<br>LONGCARD; |
| Clarion:     | AToUL(*cstring),ulong,raw,name('_atoul')      |

## Integer Math

---

|             |                                                                                   |
|-------------|-----------------------------------------------------------------------------------|
| <b>abs</b>  | Integer absolute value.                                                           |
| C++:        | int abs(int _num)                                                                 |
| Modula-2:   | abs(_num: INTEGER): INTEGER;                                                      |
| Clarion:    | API_Abs(short),short,name('_abs')<br>!Renamed to avoid conflict with Builtins.Clw |
| <b>labs</b> | Long integer absolute value.                                                      |
| C++:        | long labs(long _j)                                                                |
| Modula-2:   | labs(_i: LONGINT): LONGINT;                                                       |
| Clarion:    | LAbs(long),long,name('_labs')                                                     |

## Char Type Functions

---

The following functions have only been tested when implemented as functions. We do not advise defining `_CT_MTF` to implement the functions as macros.

|                |                                       |
|----------------|---------------------------------------|
| <b>toupper</b> | Test and convert if lowercase.        |
| C++:           | int toupper(int c)                    |
| Modula-2:      | toupper(c: INTEGER):INTEGER;          |
| Clarion:       | ToUpper(short),short,name('_toupper') |

|                |                                                                                           |
|----------------|-------------------------------------------------------------------------------------------|
| <b>tolower</b> | Test and convert if uppercase.                                                            |
| C++:           | int tolower(int c)                                                                        |
| Modula-2:      | tolower(c: INTEGER): INTEGER;                                                             |
| Clarion:       | ToLower(short),short,name('_tolower')                                                     |
| <b>isalpha</b> | Alphabetic test function.                                                                 |
| C++:           | int isalpha(int c)                                                                        |
| Modula-2:      | isalpha(c: INTEGER): INTEGER;                                                             |
| Clarion:       | API_IsAlpha(short),short,name('_isalpha')<br>!Renamed to avoid conflict with Builtins.Clw |
| <b>islower</b> | Lower case test function.                                                                 |
| C++:           | int islower(int c)                                                                        |
| Modula-2:      | islower(c: INTEGER): INTEGER;                                                             |
| Clarion:       | API_IsLower(short),short,name('_islower')<br>!Renamed to avoid conflict with Builtins.Clw |
| <b>isupper</b> | Upper case test function.                                                                 |
| C++:           | int isupper(int c)                                                                        |
| Modula-2:      | isupper(c: INTEGER): INTEGER;                                                             |
| Clarion:       | API_IsUpper(short),short,name('_isupper')<br>!Renamed to avoid conflict with Builtins.Clw |
| <b>isascii</b> | ASCII test function.                                                                      |
| C++:           | int isascii(int c)                                                                        |
| Modula-2:      | isascii(c: INTEGER): INTEGER;                                                             |
| Clarion:       | IsAscii(short),short,name('_isascii')                                                     |
| <b>isctrl</b>  | Control character test function.                                                          |
| C++:           | int isctrl(int c)                                                                         |
| Modula-2:      | isctrl(c: INTEGER): INTEGER;                                                              |

|                 |                                          |
|-----------------|------------------------------------------|
| Clarion:        | IsCntrl(short),short,name('_iscntrl')    |
| <b>isdigit</b>  | Numerics test function.                  |
| C++:            | int isdigit(int c)                       |
| Modula-2:       | isdigit(c: INTEGER): INTEGER;            |
| Clarion:        | IsDigit(short),short,name('_isdigit')    |
| <b>isgraph</b>  | Printable except space test function.    |
| C++:            | int isgraph(int c)                       |
| Modula-2:       | isgraph(c: INTEGER): INTEGER;            |
| Clarion:        | IsGraph(short),short,name('_isgraph')    |
| <b>isprint</b>  | Printable including space test function. |
| C++:            | int isprint(int c)                       |
| Modula-2:       | isprint(c: INTEGER): INTEGER;            |
| Clarion:        | IsPrint(short),short,name('_isprint')    |
| <b>ispunct</b>  | Punctuation character test function.     |
| C++:            | int ispunc(int c)                        |
| Modula-2:       | ispunc(c: INTEGER): INTEGER;             |
| Clarion:        | IsPunct(short),short,name('_ispunct')    |
| <b>isspace</b>  | Whitespace test function.                |
| C++:            | int isspace(int c)                       |
| Modula-2:       | isspace(c: INTEGER): INTEGER;            |
| Clarion:        | IsSpace(short),short,name('_isspace')    |
| <b>isxdigit</b> | Hex digit test function.                 |
| C++:            | int isxdigit(int c)                      |
| Modula-2:       | isxdigit(c: INTEGER): INTEGER;           |
| Clarion:        | IsXDigit(short),short,name('_isxdigit')  |

## Utility Functions

---

|              |                                              |
|--------------|----------------------------------------------|
| <b>rand</b>  | Return pseudorandom integer.                 |
| C++:         | int rand(void)                               |
| Modula-2:    | rand(): INTEGER;                             |
| Clarion:     | Rand(),short,name('_rand')                   |
| <b>srand</b> | Set pseudorandom seed with specified number. |
| C++:         | void srand(unsigned _seed)                   |
| Modula-2:    | srand(_seed: CARDINAL);                      |
| Clarion:     | SRand(ushort),name('_srand')                 |

## String Functions

---

|                |                                                       |
|----------------|-------------------------------------------------------|
| <b>strcat</b>  | Concatenate two strings.                              |
| C++:           | char *strcat(char *_dest, const char *_source)        |
| Modula-2:      | <i>Not available</i>                                  |
| Clarion:       | StrCat(*cstring,*cstring),cstring,raw,name('_strcat') |
| <b>strcmp</b>  | Compare two strings.                                  |
| C++:           | int strcmp(const char *_s1, const char *_s2)          |
| Modula-2:      | <i>Not available</i>                                  |
| Clarion:       | StrCmp(*cstring,*cstring),short,raw,name('_strcmp')   |
| <b>chrncmp</b> |                                                       |
| C++:           | int chrncmp(char _c1, char _c2)                       |
| Modula-2:      | chrncmp(_c1,_c2: CHAR): INTEGER;                      |
| Clarion:       | ChrCmp(byte,byte),short,name('_chrncmp')              |
| <b>strequ</b>  |                                                       |
| C++:           | int strequ(const char *_s1, const char *_s2)          |

|                 |                                                              |
|-----------------|--------------------------------------------------------------|
| Modula-2:       | <i>Not available</i>                                         |
| Clarion:        | StrEqu(*cstring, *cstring), short, raw, name('_strequ')      |
| <b>strcpy</b>   | Copy one string to another, return destination address.      |
| C++:            | char *strcpy(char *_dest, const char *_source)               |
| Modula-2:       | <i>Not available</i>                                         |
| Clarion:        | StrCpy(*cstring, *cstring), cstring, raw,   name('_strcpy')  |
| <b>strlen</b>   | Return string length.                                        |
| C++:            | unsigned strlen(const char *_s)                              |
| Modula-2:       | strlen(VAR _s: ARRAY OF CHAR): CARDINAL;                     |
| Clarion:        | StrLen(*cstring), ushort, raw, name('_strlen')               |
| <b>strchr</b>   | Find character in string.                                    |
| C++:            | char *strchr(const char *_s, int _c)                         |
| Modula-2:       | <i>Not available</i>                                         |
| Clarion:        | StrChr(*cstring, short), cstring, raw, name('_strchr')       |
| <b>strcspn</b>  | Finds one of a set of characters in string.                  |
| C++:            | unsigned strcspn(const char *_s1, const char *_s2)           |
| Modula-2:       | <i>Not available</i>                                         |
| Clarion:        | StrCSpn(*cstring, *cstring), ushort, raw,   name('_strcspn') |
| <b>strerror</b> | Concatenate user error to last system error message.         |
| C++:            | char * strerror(int _errno)                                  |
| Modula-2:       | <i>Not available</i>                                         |
| Clarion:        | StrError(short), cstring, raw, name('_strerror')             |

|                |                                                             |
|----------------|-------------------------------------------------------------|
| <b>strspn</b>  | Find first character with no match in given character set.  |
| C++:           | unsigned strspn(const char *_s1, const char *_s2)           |
| Modula-2:      | <i>Not available</i>                                        |
| Clarion:       | StrSpn(*cstring,*cstring),ushort,raw,name('_strspn')        |
| <b>strstr</b>  | Find first occurrence of substring in a string.             |
| C++:           | char *strstr(const char *_s1, const char *_s2)              |
| Modula-2:      | <i>Not available</i>                                        |
| Clarion:       | StrStr(*cstring,*cstring),cstring,raw,name('_strstr')       |
| <b>strtok</b>  | Find next token in string.                                  |
| C++:           | char *strtok(char *_s1, const char *_s2)                    |
| Modula-2:      | <i>Not available</i>                                        |
| Clarion:       | StrTok(*cstring,*cstring),cstring,raw,name('_strtok')       |
| <b>strpbrk</b> | Find first occurrence of character.                         |
| C++:           | char *strpbrk(const char *_s1, const char *_s2)             |
| Modula-2:      | <i>Not available</i>                                        |
| Clarion:       | StrPBrk(*cstring,*cstring), cstring, raw,  name('_strpbrk') |
| <b>strrchr</b> | Find last occurrence of character.                          |
| C++:           | char *strrchr(const char *_s, int _c)                       |
| Modula-2:      | <i>Not available</i>                                        |
| Clarion:       | StrRChr(*cstring,short),cstring,raw,name('_strrchr')        |
| <b>strlwr</b>  | Convert to lower case.                                      |
| C++:           | char *strlwr(char *_s)                                      |
| Modula-2:      | <i>Not available</i>                                        |
| Clarion:       | StrLwr(*cstring),cstring,raw,name('_strlwr')                |

|                |                                                                      |
|----------------|----------------------------------------------------------------------|
| <b>strupr</b>  | Convert to upper case.                                               |
| C++:           | char *strupr(char *_s)                                               |
| Modula-2:      | <i>Not available</i>                                                 |
| Clarion:       | StrUpr(*cstring),cstring,raw,name('_strupr')                         |
| <b>strdup</b>  | Duplicate string.                                                    |
| C++:           | char *strdup(const char *_s)                                         |
| Modula-2:      | <i>Not available</i>                                                 |
| Clarion:       | StrDup(*cstring),cstring,raw,name('_strdup')                         |
| <b>strrev</b>  | Reverse string.                                                      |
| C++:           | char *strrev(char *_s)                                               |
| Modula-2:      | <i>Not available</i>                                                 |
| Clarion:       | StrRev(*cstring),cstring,raw,name('_strrev')                         |
| <b>strncat</b> | Concatenate n characters.                                            |
| C++:           | char *strncat(char *_dest, const char *_source, unsigned _n)         |
| Modula-2:      | <i>Not available</i>                                                 |
| Clarion:       | StrNCat(*cstring, *cstring, ushort), cstring, raw,  name('_strncat') |
| <b>strncmp</b> | Compare n characters.                                                |
| C++:           | int strncmp(const char *_s1, const char *_s2, unsigned _n)           |
| Modula-2:      | <i>Not available</i>                                                 |
| Clarion:       | StrNCmp(*cstring, *cstring, ushort), short, raw,  name('_strncmp')   |
| <b>strncpy</b> | Copy n characters.                                                   |
| C++:           | char * strncpy(char *_dest, const char *_source, unsigned _n)        |

|                 |                                                                         |
|-----------------|-------------------------------------------------------------------------|
| Modula-2:       | <i>Not available</i>                                                    |
| Clarion:        | StrNCpy(*cstring, *cstring, ushort), cstring, raw, <br>name('_strncpy') |
| <b>strnicmp</b> | Compare n characters regardless of case.                                |
| C++:            | int stricmp(const char *_s1, const char *_s2,<br>unsigned _n)           |
| Modula-2:       | <i>Not available</i>                                                    |
| Clarion:        | StrNICmp(*cstring, *cstring, ushort), short, raw, <br>name('_strnicmp') |

## Low-Level File Manipulation

---

|               |                                                                                                                                                           |
|---------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>access</b> | Checks whether the file (or directory) specified by the path parameter exists, and (if not a directory) whether it can be accessed in the specified mode. |
| C++:          | int _access(const char *path, int access)                                                                                                                 |
| Modula-2:     | _access(VAR path: ARRAY OF CHAR; access:<br>INTEGER): INTEGER;                                                                                            |
| Clarion:      | Access(*cstring,short),short,raw,name('_access')                                                                                                          |
| <b>chmod</b>  | Set file's access mode.                                                                                                                                   |
| C++:          | int _chmod(const char *path, int mode)                                                                                                                    |
| Modula-2:     | _chmod(VAR path: ARRAY OF CHAR; mode:<br>INTEGER): INTEGER;                                                                                               |
| Clarion:      | ChMod(*cstring,short),short,raw,name('_chmod')                                                                                                            |
| <b>remove</b> | Deletes the file specified by the path parameter.                                                                                                         |
| C++:          | int _remove(const char *_path)                                                                                                                            |
| Modula-2:     | _remove(VAR _path: ARRAY OF CHAR):<br>INTEGER;                                                                                                            |
| Clarion:      | API_Remove(*cstring),short,raw,name('_remove')<br>!Renamed to avoid conflict with Builtins.Clw                                                            |

|                |                                                                                                                            |
|----------------|----------------------------------------------------------------------------------------------------------------------------|
| <b>rename</b>  | Changes the name of the file or directory specified by the oldname parameter.                                              |
| C++:           | <code>int _rename(const char *_oldname, const char *_newname)</code>                                                       |
| Modula-2:      | <code>_rename(VAR _oldname, VAR _newname: ARRAY OF CHAR): INTEGER;</code>                                                  |
| Clarion:       | <code>API_Rename(*cstring, *cstring), short, raw,   name('_rename')</code><br>!Renamed to avoid conflict with Builtins.Clw |
| <b>fnmerge</b> | Builds a complete path name from its component parts -- drive, directory, filename, and extension.                         |
| C++:           | <code>void _fnmerge(char *_path, const char *_drive, const char *_dir, const char *_name, const char *_ext)</code>         |
| Modula-2:      | <code>_fnmerge(VAR _path, VAR _drive, VAR _dir, VAR _name, VAR _ext: ARRAY OF CHAR);</code>                                |
| Clarion:       | <code>FnMerge(*cstring, *cstring, *cstring, *cstring,   *cstring), raw, name('_fnmerge')</code>                            |
| <b>fnsplit</b> | This function breaks a complete path name into its component parts -- drive, directory, filename, and extension.           |
| C++:           | <code>int _fnsplit(const char *_path, char *_drive, char *_dir, char *_name, char *_ext)</code>                            |
| Modula-2:      | <code>_fnsplit(VAR _path, VAR _drive, VAR _dir, VAR _name, VAR _ext: ARRAY OF CHAR): INTEGER;</code>                       |
| Clarion:       | <code>FnSplit(*cstring, *cstring, *cstring, *cstring,   *cstring), short, raw, name('_fnsplit')</code>                     |
| <b>mkdir</b>   | Creates a new directory with the name passed in the path parameter.                                                        |
| C++:           | <code>int _mkdir(const char *_path)</code>                                                                                 |
| Modula-2:      | <code>_mkdir(VAR _path: ARRAY OF CHAR): INTEGER;</code>                                                                    |
| Clarion:       | <code>MkDir(*cstring), short, raw, name('_mkdir')</code>                                                                   |

|                  |                                                               |
|------------------|---------------------------------------------------------------|
| <b>rmdir</b>     | removes the directory specified in the path parameter.        |
| C++:             | int _rmdir(const char *_path)                                 |
| Modula-2:        | _rmdir(VAR _path: ARRAY OF CHAR):INTEGER;                     |
| Clarion:         | RmDir(*cstring),short,raw,name('_rmdir')                      |
| <b>getcurdir</b> | Get current directory.                                        |
| C++:             | int _getcurdir(int drive, char *_buf)                         |
| Modula-2:        | _getcurdir(drive: INTEGER; VAR _buf: ARRAY OF CHAR): INTEGER; |
| Clarion:         | GetCurDir(short, *cstring), short, raw, name('_getcurdir')    |
| <b>chdir</b>     | Change directory.                                             |
| C++:             | int _chdir(const char *_path)                                 |
| Modula-2:        | _chdir(VAR _path: ARRAY OF CHAR): INTEGER;                    |
| Clarion:         | ChDir(*cstring),short,raw,name('_chdir')                      |
| <b>getdisk</b>   | Current disk drive number.                                    |
| C++:             | int _getdisk(void)                                            |
| Modula-2:        | _getdisk(): INTEGER;                                          |
| Clarion:         | GetDisk(),short,name('_getdisk')                              |
| <b>setdisk</b>   | Set current disk drive number.                                |
| C++:             | int _setdisk(int _drive)                                      |
| Modula-2:        | _setdisk(_drive: INTEGER): INTEGER;                           |
| Clarion:         | SetDisk(short),short,name('_setdisk')                         |



# PART III



# TEMPLATE LANGUAGE PROGRAMMING



# **15 - INTRODUCTION TO TEMPLATE LANGUAGE**

## **Template Language Overview**

Clarion for Windows' Template Language is a flexible script language complete with control structures, user interface elements, variables, file I/O, and more. The Template Language “drives” the Application Generator both at application design time and during source code generation.

- During application design, the programmer is asked for specific information about the application being generated. These prompts for information come directly from the templates.
- During source code generation, the template is in control of the source code statements generated for each procedure in the application, and also controls what source files receive the generated code.

This process makes the Templates completely in control of the Application Generator. The benefit to the programmer of this is the complete flexibility to generate code that is directly suited to the programmer's needs.

## **What Templates Are**

---

A template is a complete set of instructions, both Template and “target” language statements, which the Application Generator uses to process the programmer's input for application customizations then generate “target” language (usually, but not limited to, Clarion language) source code.

Clarion's templates are completely reusable. They generate only the exact code required for each specific instance of its use; they do not inherit unused methods. The templates are also polymorphic, since the programmer specifies the features and functions of each template that are required for the procedure. This means one template can generate different functionality based upon the programmer's desires.

Some of the most important aspects of template functionality supported by the Template Language include:

- Support for controls (#PROMPT) that gather input from the developer, storing that input in user-defined template variables (symbols).
- Pre-defined template variables (Built-in Symbols) containing information from the data dictionary and Clarion for Windows' application development environment.
- Specialized #PROMPT entry types, which give the programmer a list of appropriate choices for such things as data file or key selection.

- Unconditional and conditional control structures (#FOR, #LOOP, #IF, #CASE) which branch source generation execution based on an expression or the contents of a symbol (variable). This allows the Application Generator to generate only the exact source code needed to produce the programmer's desired functionality in their application.
- Statements (#EMBED) that define specific points where the developer can insert (embed) or *not* insert their own source code to further customize their application.
- Support for code templates (#CODE), control templates (#CONTROL), and extension templates (#EXTENSION) that add their specific (extended) functionality to any procedure template. This makes any procedure type polymorphic, in that, the procedure can include functionality normally performed by other types of procedures.

Template code is contained in one or more ASCII files (\*.TPL or \*.TPW) which the Application Generator pre-compiles and incorporates into the REGISTRY.TRF file. It is this template registry file that the Application Generator uses during application design.

Once in the registry, the template code is completely reusable from application to application. It generates custom source code for each application based upon the application's data dictionary and the options selected by the programmer while working with the Application Generator.

The programmer can customize the templates in the registry (or in the \*.TP\* files) to fit their own specific standard design requirements. This means that each procedure template can be designed to appear exactly as the programmer requires as a starting point for their applications. Multiple "default" starting points can be created, so the programmer can have a choice of starting point designs for each procedure type.

When the programmer has customized the template source (\*.TP\* file), the Application Generator automatically updates the registry. When the programmer has customized the registry, the template source files can be re-generated from the registry, if necessary.

The Application Generator always makes a *copy* of the template, as stored in the registry, when creating a procedure or first populating a procedure with a code, control, or extension template. Once this copy is made, the programmer further customizes it to produce exactly the functionality required by the application for that procedure.

The template language can generate more than source code: it can even be used to create add-in utilities (see #UTILITY).

## Template Types

---

There are four main types of templates: procedure, code, control, and extension templates.

- Procedure templates (`#PROCEDURE`) generate procedures in an application. This is the choice you make when asked to choose the starting point for a “ToDo” procedure in the Application Generator.
- Code templates (`#CODE`) generate executable code into a specific embed point. The developer can only insert them at an embed point within a procedure. A list of the available code templates appears from which to choose.
- Control templates (`#CONTROL`) place a related set (one or more) of controls on a procedure’s window and generate the executable source code into the procedure’s embed points to provide the controls’ standard functionality.
- Extension templates (`#EXTENSION`) generate executable source code into one or more embed points to add specific functionality to a procedure that is not “tied” to any window control.

## What Templates Do

---

The template code files contain template language statements and standard “target” language source code which the Application Generator places in your generated source code files. They also contain the prompts for the Application Generator which determine the standard customizations the developer can make to the generated code.

The programmer’s response (or lack of) to the prompts “drives” the control statements that process the template language code, and produces the logic that generates the source code. The templates also contain control statements which instruct the Application Generator how to process the standard code. The function of a template is to generate the “target” language source code, customized per the programmer’s response to the prompts and design of the window or report.

There are some lines of code from templates that are inserted directly into your generated source code. For example, if you accept a default Copy command menu item in your application window, the following code is inserted in your generated source exactly as it appears in the template file:

```
ITEM('&Copy'),USE(?Copy),STD(STD:Copy),MSG('Copy item to Windows clipboard')
```

Some of the standard code in the template is a mix of “target” (Clarion) language statements and template language statements. For example, when the contents of a template variable (symbol) needs to be inserted in the generated source code, the Application Generator expands the symbol to the

value the application will use, as it generates the source code for the application. Within the template code, the percent sign (%) identifies a variable (symbol). In the example below, the Application Generator will fill in the field `equate` label for the control as it writes the source code file, substituting it for the `%Control` variable:

```
SELECT(%Control)
```

To support customizing the template starting point at design time, Clarion's template language provides prompt statements that generate the template's user interface, so that the Application Generator can query the developer for the information needed to customize the application. The basic interface consists of command buttons, check boxes, radio buttons, and entry controls placed on the Procedure Properties dialog. These statements can also create custom dialog boxes to gather input from the developer. While working with the Application Generator, therefore, some of the dialogs and other interface elements the developer sees are not part of the Application Generator—rather they are produced by the template.

For example, the following statement displays a file selection dialog from the application's data dictionary, then stores the programmer's choice for a data file in a variable (symbol) called `%MyFile`:

```
#PROMPT('Pick a file',FILE),%MyFile
```

It makes no difference what the programmer names the files and fields, nor what database driver is selected. The programmer picks them from a file selection dialog.

The template also contains control structures to instruct the Application Generator on how to generate the code( such as, `#IF`, `#LOOP`, `#CASE`). These control statements work in the same manner as Clarion language control structures.

## Pre-Processing and Source Code Generation

---

Before allowing you to create an application using the templates, the Application Generator pre-processes the template code (.TPL and .TPW) files. The Application Generator verifies the registry is up to date by testing the time stamps and file sizes of all the template source code files.

The Application Generator utilizes the templates as stored in binary form in the registry file, as it gathers customizations from the developer with the prompts and dialogs available through the Procedure Properties dialog. The Application Generator stores the template starting point for each procedure and the customization from the programmer in the .APP file.

At source code generation time, the Application Generator processes the application's procedures as stored in the .APP file against the template, a

second time. Some of the more important steps it uses to produce the source code are:

- It executes the template language control statements to process the template and the procedure's customizations in the correct order.
- It resolves the template symbols—both built-in and user-defined.
- It creates the source code files and writes the source code as generated by the template, line by line, including the previously evaluated symbols.
- It evaluates embed points and writes the source code, as embedded by the developer and stored in the .APP file, in the correct location within the generated source code.

## Embed Points

---

One of the most important template language statements is `#EMBED`, which defines an embed point. These extend the structure and functionality of the procedure template by allowing the programmer to add their own custom code. The embed points indicate “targets” at which the developer can add their own custom code to the generated source. These are also the “targets” for the source code generated by control and extension templates.

Each procedure template allows for a certain number of default points at which embeds are allowed. These are typically points which coincide with messages (events) from the operating environment (Windows), such as when the end user moves focus from or to a field. The template programmer can add to, or subtract from, the list.

When the developer customizes the template, pressing the Embeds button in the Procedure Properties dialog provides access to all the embed points available in a procedure. The Actions popup menu selection in the Window Formatter also provides access to the embed points for a specific control.

The developer adds custom code—either hand coded from scratch in the editor, or created with a code template—at the embed point. The embed points are also the points into which control templates and extension templates generate executable code to support their functionality.

The Application Generator stores the embed point's code (no matter what its origination) in the .APP file. At code generation time, the Application Generator processes the template, producing source; when it reaches an embed point, it places the developer's code, line by line, into the generated source code document.

## Template Prompts

---

Input Validation Statements and Prompt Entry types place controls on the Procedure Properties window or Actions dialog which the developer sees when using the template to design an application. These range from a simple string telling the Developer what to do (#DISPLAY), to command buttons, check boxes, or radio buttons. There are also specialized entry types which provide the programmer a list of choices for input, such as the data fields in the dictionary.

Standard Windows controls can be used to get information from the programmer on the Procedure Properties window, the Actions dialog, or custom prompt dialogs. The common control types—entry field, check box, radio button, and drop-down list—are all directly supported via the #PROMPT statement.

#PROMPT places the prompt, the input control, and the symbol in a single statement. The general format is the #PROMPT keyword, the string to display the actual prompt, a variable type for the symbol, then the symbol or variable name. The Application Generator places the prompt and the control in the Procedure Properties or Actions dialog (depending on whether the prompt comes from a the procedure template or a code, control, or extension template). When the developer fills the control with a value, then closes the dialog, the symbol holds the value.

The #BUTTON statement provides additional “space” for developer input when there is more developer input required than can fit in the one dialog. This places a button in the dialog, which displays an additional custom dialog when pressed. The additional dialogs are called “prompt pages.”

#ENABLE allows prompts to be conditionally enabled based on the programmer's response to some other prompt. #BOXED supports logical grouping of related prompts. Once the programmer has input data into a prompt, the #VALIDATE statement allows the template to check its validity.

These tools provide a wide range of flexibility in the type of information a template can ask the programmer to provide. They also provide multiple ways to expedite the programmer's job, by providing “pick-lists” from which the programmer may choose wherever appropriate.

## Data Dictionary Interface

---

The templates use information from the Data Dictionary extensively to generate code specifically for the declared database. There are several symbols that specifically give the templates access to all the declarations: %File, %Field, %Key, and %Relation. These, and all the symbols related to them, give the templates access to all the information in the Data Dictionary.

Pay special attention to the %FileUserOptions, %FieldUserOptions, %KeyUserOptions, and %RelationUserOptions symbols. These are the symbols that contain the values the user enters in the **User Options** text control on the **Options** tab of the **File Properties**, **Field Properties**, **Key Properties**, and **Relation Properties** dialogs. This can be a powerful tool to customize any output from the Data Dictionary.

The best way to use these %UserOptions symbols is to set them up so the user enters their custom preferences which your template supports in the form of attributes with parameters, with each attribute separated by a comma. This gives them the same appearance as Clarion language data structure attributes. By doing this, you can use the EXTRACT built-in template procedure to get the value from the user. For example, if the user enters the following in a **User Options** for a field:

```
MYCUSTOMOPTION(On)
```

The template code can parse this using EXTRACT:

```
#IF(EXTRACT(%FieldUserOptions,'MYCUSTOMOPTION',1) = On)
 #!Do Something related to this option being turned on
#ENDIF
```

This is a very powerful tool, which allows for infinite flexibility in the way your custom templates generate source code.

# Template Structure

## Template Source Format

---

The structure of the ASCII template source file is different than the structure of a Clarion source file. To read the ASCII source for a template, start out with the following guidelines:

- Any statement beginning with a pound symbol (#) identifies a template language statement.
- A percent sign (%) before an item within any statement (template or “target” language) identifies a template symbol (variable), which the Application Generator processes at code generation time.
- Any statement that begins without the pound (#) or percent (%) is a “target” language statement which is written directly into a source code file.

The template files are organized by code sections that terminate with the beginning of the next section or the end of the file. The template code generally divides into ten sections.

- **#TEMPLATE** begins a template set (template class). This is the first statement in the template set (required) which identifies the template set for the registry.
- **#APPLICATION** begins the source generation control section. This is the section of the template that controls the “target” language code output to source files, ready to compile. One registered template set must have a **#APPLICATION** section.
- **#PROGRAM** begins the global section of the generated source code, the main program module. One registered template set must have a **#PROGRAM** section.
- **#MODULE** begins a template section that generates the beginning code for a source code module other than the global (program) file. One registered template set must have a **#MODULE** section.
- **#PROCEDURE** begins a procedure template. This is the fundamental “target” language procedure generation template.
- **#GROUP** begins a reusable statement group containing code which may be **#INSERTed** into any other section of the template. This is the equivalent of a template language procedure.
- **#CODE** begins a code template section which generates executable code into a specific embed point. The developer can only insert them at an embed point within a procedure. A list of the available code templates appears from which to choose.

- `#CONTROL` begins a control template. Control templates place a related set (one or more) of controls on a procedure's window and generate the executable source code into embed points that provides the controls' standard functionality.
- `#EXTENSION` begins an extension template. Extension templates generate executable source code into one or more embed points of a procedure to add specific functionality to the procedure that is not "tied" to any window control.
- `#UTILITY` begins a utility execution section. This is an optional section of the template that performs a utility function, such as cross-reference or documentation generation. This is similar to `#APPLICATION` in that it generates output to ASCII files.

A template set must have a `#TEMPLATE` section to name the set for registration in the `REGISTRY.TRF` template registry file. At least one registered template set must have `#APPLICATION`, `#PROGRAM`, and `#MODULE` sections.

## The Template Registry File

---

The Template Registry file (`REGISTRY.TRF`) is a specialized data repository which stores template code and defaults in binary form. All the template elements available in the Application Generator come from the registry. As you add elements from the template into your application, the Application Generator retrieves the code from the registry then stores it along with your customizations, in the `.APP` file.

Storing the templates in a binary registry provides these advantages:

- Quick design-time performance.
- The ability to update the defaults in the registry using standard application development tools (such as the Window Formatter). For example, you can modify a procedure template's default window without writing template source code.

The sources for the `REGISTRY.TRF` are the template code files (`.TPL` and `.TPW`) which are installed in the `TEMPLATE` subdirectory. The Application Generator can read and register `.TPL` files, adding it to the template registry tree. The `.TPW` files usually contain additional procedure or code template source, which is processed along with the `.TPL` file by the `#INCLUDE` statement in the `.TPL` file. This allows the template author to logically separate disparate template components.

The default template file for Clarion for Windows is `CW.TPL`. This file uses the `#INCLUDE` statement to specify processing the the other `.TPW` files which appear in the `\CW\TEMPLATE` directory.

## Customizing Default Templates

---

There are two methods for customizing the templates:

- You can edit the template source code in the .TPL and .TPW files.

*It is always a good idea to make a backup copy before making any modifications to the shipping templates.*

When directly editing the template source code, you can change the type of source code it generates, or the logic it uses to generate the code. This is how you can make your templates generate source code the way you would write it if you were hand-coding the application.

You can also extend the functionality of the templates by adding your own features. For example, you may want to add prompts to each procedure template that allow you to generate a “comment block” at the beginning of each procedure containing procedure maintenance comments from the programmer maintaining the application.

Adding the following code to the end of any existing template set accomplishes this modification:

```
#EXTENSION(CommentBlock,'Add a comment block to the procedure'),PROCEDURE
 #PROMPT('Comment Line',@S70),%MyComment,MULTI('Programmer Comments')
#ATSTART
 #FOR(%MyComment)
 !%MyComment
 #ENDFOR
#ENDAT
```

This code adds an extension template that is available for any procedure in the application. When you design your procedure, add the CommentBlock extension template to the procedure, then add comments to the Comment Line prompt each time you modify the procedure. At source generation time, each comment line will appear following an exclamation point (!). The block of comments appears in the code just before the PROCEDURE statement.

If you want this extension to be used in all the procedures you write, go into the Template Registry and add the extension to all the default procedures for each procedure template. This way, you can make sure it is always used, and you can even place its prompts on the Procedure Properties dialog by checking the Show on Properties box as you add the extension to the procedure template.

Once you make the changes, either choose the **Setup ► Template Registry** menu selection, open an existing application, or create a new application. Make sure the Re-register When Changed box is checked in the Registry Options dialog. The Application Generator automatically pre-processes the

templates to update the registry when you have made changes to the template code files.

- You can add to or edit the default user interface procedure template elements—such as the standard window designs and report layouts, or your standard global and local data variables—using the Template Registry.

When you highlight a procedure template in the Template Registry and press the Properties button, the Procedure Properties dialog appears, without all the custom prompts you would normally see when developing an application. Any button which is not dimmed in the Template Registry is available to you to create the default starting point for the procedure.

You can set up the procedure for the starting point that will get you furthest toward a complete procedure while requiring the least amount of customization from you at application design time. If the procedure allows it, you may use the window and report formatters, or define additional data, by pressing the appropriate buttons.

Once you've customized your template registry, you can also export your customizations to template source code files. This is useful for sharing your customizations with other developers.

To update the template source code with the customizations made in the Template Registry, press the Regenerate button in the Template Properties dialog. This updates the .TPL and .TPW files with the changes made.

## Adding New Template Sets

---

Adding another set of templates, whether from a third-party vendor or templates you have written yourself, is a very simple process. There is only one requirement for the new template set; a #TEMPLATE statement to identify the set for the template registry with a FAMILY attribute. Of course, it also needs to have the specific procedure, code, control, and extension templates to add to the template registry.

For example, the following code is completely valid as a template set with nothing else added:

```
#TEMPLATE(PersonalAddOns,'My personal Template set'),FAMILY('ABC','Clarion')
#CODE(ChangeProperty,'Change control property')
 #PROMPT('Control to change',CONTROL,%MyField,REQ
 #PROMPT('Property to change',@S20),%MyProperty,REQ
 #PROMPT('New Value',@S20),%MyValue,REQ
%MyField{%MyProperty} = '%MyValue' #<!Change the %MyProperty of %MyField
```

When you register this template set, it will appear in the template registry as Class PersonalAddOns containing just the ChangeProperty code template.

Once a template set is registered in the template registry, all its components are completely available to the programmer for their application development, along with all the components of all other registered template sets. This allows the programmer the flexibility to “mix-and-match” their components during development.

For example, the programmer could create a procedure from a procedure template in the standard Clarion template set, populate it with a control template from a third-party vendor, insert a code template into an embed point from another third-party vendor, then add an extension template from their own personally written template set. At source generation time, all these separate components come together to create a fully functional procedure that performs all the tasks required by the programmer (and nothing else). This is the real power behind Clarion's Template-oriented programming!

# 16 - TEMPLATE ORGANIZATION

## Template Code Sections

### #TEMPLATE (begin template set)

```
#TEMPLATE(name, description) [, PRIVATE] [, FAMILY(sets)]
```

|                    |                                                                                                                                                                                                                                       |
|--------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>#TEMPLATE</b>   | Begins the Template set.                                                                                                                                                                                                              |
| <i>name</i>        | The name of the Template set which uniquely identifies it for the Template Registry and Template Language statements. This must be a valid Clarion label.                                                                             |
| <i>description</i> | A string constant describing the Template set for the Template Registry and Application Generator.                                                                                                                                    |
| <b>PRIVATE</b>     | Prevents re-generation from the Template Registry.                                                                                                                                                                                    |
| <b>FAMILY</b>      | Names the other Template <i>sets</i> in which the #TEMPLATE is valid for use. There may be multiple FAMILY attributes on the same #TEMPLATE statement. If omitted, the #TEMPLATE is valid for use only with the Clarion template set. |
| <i>sets</i>        | A string constant containing a comma delimited list of the <i>names</i> of other Template sets in which the #TEMPLATE is valid for use.                                                                                               |

The **#TEMPLATE** statement marks the beginning of a Template set. This should be the first non-comment statement in the Template file. The Template Registry allows multiple Template sets to be registered for the Application Generator.

Each Template Code Section (#APPLICATION, #PROGRAM, #MODULE, #PROCEDURE, #CONTROL, #CODE, #EXTENSION, and #GROUP) within a Template is uniquely identified by its #TEMPLATE statement's *name* and the name of the section. This allows different Template sets to contain Template Code Sections with names that duplicate those in other Template sets without ambiguity, and allows the programmer to concurrently use Template sets from multiple sources to generate applications.

Example:

```
#TEMPLATE(SampleTemplate,'This is a sample Template'),PRIVATE,FAMILY('ABC')
#include('FileTwo.TPW')
#include('FileThree.TPW')
```

## #SYSTEM (template registration and load)

---

### #SYSTEM

The **#SYSTEM** statement marks the beginning of a section of Template code which executes when the Template set is registered or loaded from the template registry. The section is terminated by the next Template Code Section (**#PROGRAM**, **#MODULE**, **#PROCEDURE**, **#CONTROL**, **#CODE**, **#EXTENSION**, **#UTILITY**, or **#GROUP**) statement. Any **#PROMPT** statements in a **#SYSTEM** section appear on the Application Options dialog, once the Template Registry has been loaded (opened an .APP file).

Example:

```
#SYSTEM
#TAB('Wizards')
#PROMPT('Use Toolbar mode for Wizards',CHECK),%UseToolBarFor Wizard
#ENDTAB
#DECLARE(%GlobalSystemSymbol)
```

## #APPLICATION (source generation control section)

```
#APPLICATION(description) [, HLP(helpid)] [APPLICATION([child(chain)]
```

|                     |                                                                                                                                                                                 |
|---------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>#APPLICATION</b> | Begins source generation control section.                                                                                                                                       |
| <i>description</i>  | A string constant describing the application section.                                                                                                                           |
| <b>HLP</b>          | Specifies on-line help is available.                                                                                                                                            |
| <i>helpid</i>       | A string constant containing the identifier to access the Help system. This may be either a Help keyword or “context string.”                                                   |
| <b>APPLICATION</b>  | Tells the Application Generator to automatically place the named) <i>child</i> template on every procedure.                                                                     |
| <i>child(chain)</i> | The name of a #EXTENSION with the PROCEDURE attribute to automatically populate into every generated procedure when the #EXTENSION with the APPLICATION attribute is populated. |

The **#APPLICATION** statement marks the beginning of a source generation control section. The section is terminated by the next Template Code Section (**#PROGRAM**, **#MODULE**, **#PROCEDURE**, **#CONTROL**, **#CODE**, **#EXTENSION**, **#UTILITY**, or **#GROUP**) statement. The Template statements contained in this section control the source generation process. Only one **#APPLICATION** section is allowed in a single Template set. Actual source generation is done by the **#GENERATE** statement.

Any User-defined symbols defined in the **#APPLICATION** section are available for use in any Template Code Section that is generated. Any prompts in this section are placed on the Global Properties window and have global scope.

Example:

```
#APPLICATION('Example Application Section') #!Generate entire application
#PROMPT('Enable &Shared Files',CHECK),%SharedFiles
#PROMPT('Close Unused &Files',CHECK),%CloseFiles,DEFAULT(1)
#BUTTON('.INI File Settings')
 #PROMPT('Use .INI file',CHECK),%INIActive,DEFAULT(1)
 #ENABLE(%INIActive)
 #PROMPT('.INI File to use',DROP,'Program Name.INI|Other'),%INIFile
 #ENABLE(%INIFile='Other')
 #PROMPT('File Name',@S40),%ININame
 #ENDENABLE
 #PROMPT('Save Window Locations',CHECK),%INISaveWindow,DEFAULT(1)
 #ENDENABLE
#ENDBUTTON
#!
#!-----Global Template Declarations.
#MESSAGE('Generating ' & %Application,0)#! Open the Message Box
#DECLARE(%FilesUsed),UNIQUE,MULTI #! Label of every file used
#DECLARE(%FilePut,%FilesUsed) #! "Yes" for RI PUT used
#DECLARE(%FileDelete,%FilesUsed) #! "Yes" for RI DELETE used
```

```

#DECLARE(%ModuleFilesUsed,%Module),UNIQUE,MULTI,SAVE #!Name of file used in module
#DECLARE(%ModuleFilePut,%ModuleFilesUsed),SAVE #! "Yes" for RI PUT used
#DECLARE(%ModuleFileDelete,%ModuleFilesUsed),SAVE #! "Yes" for RI DELETE used
#DECLARE(%IniFileName) #! Used to construct INI file
#DECLARE(%ModuleProcs,%Module),MULTI,SAVE,UNIQUE #! Program MAP prototype
#DECLARE(%ModulePrototype,%ModuleProcs) #! Module MAP prototype
#DECLARE(%AccessMode) #! File open mode equate
#DECLARE(%BuildFile) #! Construction filename
#!
#!-----Initialization Code for Global User-defined Symbols.
#IF(%SharedFiles) #! IF Shared Files Enabled
 #SET(%AccessMode,'42h') #! default access 'shared'
#ELSE #! ELSE (IF NOT Shared Files ..)
 #SET(%AccessMode,'22h') #! default access 'open'
#ENDIF #! END (IF Shared Files ...)
#IF(%INIFile = 'Program Name.INI') #! IF using program.ini
 #SET(%INIFileName, %Application & '.INI') #! SET the file name
#ELSE #! ELSE (IF NOT using Program.ini)
 #SET(%INIFileName,%ININame) #! SET the file name
#ENDIF #! END (IF using program.ini)
#!
#!----- Main Source Code Generation Loop.
#DECLARE(%GlobalRegenerate) #! Flag that controls generation
#IF(~%ConditionalGenerate OR %DictionaryChanged OR %RegistryChanged)
 #SET(%GlobalRegenerate,%True) #! Generate Everything
#ELSE #! ELSE (If no global change)
 #SET(%GlobalRegenerate,%False) #! Generate changed modules only
#ENDIF #! END (IF Global Change)
#SET(%BuildFile,(%Application & '.TM$')) #! Make temp program filename
#FOR(%Module), WHERE (%Module <> %Program) #! For all member modules
 #MESSAGE('Generating Module: ' & %Module, 1) #! Post generation message
 #IF(%ModuleChanged OR %GlobalRegenerate) #! IF module to be generated
 #FREE(%ModuleProcs) #! Clear module prototypes
 #FREE(%ModuleFilesUsed) #! Clear files used
 #CREATE(%BuildFile) #! Create temp module file
 #FOR(%ModuleProcedure) #! FOR all procs in module
 #FIX(%Procedure,%ModuleProcedure) #! Fix current procedure
 #MESSAGE('Generating Procedure: ' & %Procedure, 2) #! Post generation message
 #GENERATE(%Procedure) #! Generate procedure code
 #ENDFOR #! END (For all procs in module)
 #CLOSE(%BuildFile) #! Close last temp file
 #CREATE(%Module) #! Create a module file
 #GENERATE(%Module) #! Generate module header
 #APPEND(%BuildFile) #! Append the temp mod file
 #CLOSE(%Module) #! Close the module file
 #ENDIF #! END (If module to be...)
#ENDFOR #! END (For all member modules)
#FIX(%Module,%Program) #! FIX to program module
#MESSAGE('Generating Module: ' & %Module, 1) #! Post generation message
#FREE(%ModuleProcs) #! Clear module prototypes
#FREE(%ModuleFilesUsed) #! Clear files used
#CREATE(%BuildFile) #! Create temp module file
#FOR(%ModuleProcedure) #! For all procs in module
 #FIX(%Procedure,%ModuleProcedure) #! Fix current procedure
 #MESSAGE('Generating Procedure: ' & %Procedure, 2) #! Post generation message
 #GENERATE(%Procedure) #! Generate procedure code
#ENDFOR #! EndFor all procs in module
#CLOSE() #! Close last temp file

```

See Also:

**#GENERATE**

## #PROGRAM (global area)

```
#PROGRAM(name, description [, target, extension]) [, HLP(helpid)]
```

|                    |                                                                                                                                              |
|--------------------|----------------------------------------------------------------------------------------------------------------------------------------------|
| <b>#PROGRAM</b>    | Defines the beginning of the main program module.                                                                                            |
| <i>name</i>        | The name of the #PROGRAM which identifies it for the Template Registry and Template Language statements. This must be a valid Clarion label. |
| <i>description</i> | A string constant describing the #PROGRAM section for the Template Registry and Application Generator.                                       |
| <i>target</i>      | A string constant that specifies the source language the Template generates. If omitted, it defaults to Clarion.                             |
| <i>extension</i>   | A string constant that specifies the source code file extension for the <i>target</i> . If omitted, it defaults to .CLW.                     |
| <b>HLP</b>         | Specifies on-line help is available.                                                                                                         |
| <i>helpid</i>      | A string constant containing the identifier to access the Help system. This may be either a Help keyword or “context string.”                |

The **#PROGRAM** statement defines the beginning of the main program module of the Template. The #PROGRAM section is terminated by the next Template Code Section (#MODULE, #PROCEDURE, #CONTROL, #CODE, #EXTENSION, or #GROUP) statement encountered, or the end of the file. Only one #PROGRAM section is allowed in a Template set.

#BUTTON, #PROMPT, and #DISPLAY statements are not valid within a #PROGRAM section. Global prompts go in the #APPLICATION section.

Example:

```
#PROGRAM(CLARION,'Standard Clarion Shipping Template')
PROGRAM !PROGRAM statement required
INCLUDE('Keycodes.clw')
INCLUDE('Errors.clw')
INCLUDE('Equates.clw')
```

## #MODULE (module area)

**#MODULE**( *name, description* [, *target, extension* ] ) [, **HLP**( *helpid* ) ] [, **EXTERNAL** ]

|                    |                                                                                                                                                                                                             |
|--------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>#MODULE</b>     | Begins the module section.                                                                                                                                                                                  |
| <i>name</i>        | The name of the Module which identifies it for the Template Registry and Template Language statements. This must be a valid Clarion label.                                                                  |
| <i>description</i> | A string constant describing the #MODULE section for the Template Registry and Application Generator.                                                                                                       |
| <i>target</i>      | A string constant that specifies the source language the Template generates. The word "EXTERNAL" is convention adopted to indicate an external source or object module. If omitted, it defaults to Clarion. |
| <i>extension</i>   | A string constant that specifies the source code file extension for the <i>target</i> . If omitted, it defaults to .CLW.                                                                                    |
| <b>HLP</b>         | Specifies on-line help is available.                                                                                                                                                                        |
| <i>helpid</i>      | A string constant containing the identifier to access the Help system. This may be either a Help keyword or "context string."                                                                               |
| <b>EXTERNAL</b>    | Specifies no source generates into the module.                                                                                                                                                              |

The #MODULE statement defines the beginning of the section of the template which puts data into each generated source module's data area. The #MODULE Section is terminated by the next Template Code Section (#PROGRAM, #MODULE, #PROCEDURE, #CONTROL, #CODE, #EXTENSION, or #GROUP) statement encountered, or the end of the file. A Template set may contain multiple #MODULE statements.

Code generated by a #MODULE section is (usually) placed at the beginning of a source code file generated by the Application Generator.

#BUTTON, #PROMPT, and #DISPLAY statements are not valid within a #MODULE section.

Example:

```
#MODULE(ExternalOBJ,'External .OBJ module','EXTERNAL','.OBJ'),EXTERNAL
#MODULE(ExternalLIB,'External .LIB module','EXTERNAL','.LIB'),EXTERNAL
#MODULE(GENERATED,'Clarion MEMBER module')
 MEMBER('%Program') !MEMBER statement is required
%ModuleData #!Data declarations local to the Module
```

## #PROCEDURE (begin a procedure template)

```
#PROCEDURE(name, description [, target]) [, REPORT] [, WINDOW] [, HLP(helpid)]
[, PRIMARY(message [, flag])] [, QUICK(wizard)]
```

|                    |                                                                                                                                                |
|--------------------|------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>#PROCEDURE</b>  | Begins a procedure template.                                                                                                                   |
| <i>name</i>        | The label of the procedure template. This must be a valid Clarion label.                                                                       |
| <i>description</i> | A string constant describing the procedure Template.                                                                                           |
| <i>target</i>      | A string constant that specifies the source language the template generates. If omitted, it defaults to Clarion.                               |
| <b>REPORT</b>      | Tells the Application Generator to make the Report Formatter available.                                                                        |
| <b>WINDOW</b>      | Tells the Application Generator to make the Window Formatter available.                                                                        |
| <b>HLP</b>         | Specifies on-line help is available.                                                                                                           |
| <i>helpid</i>      | A string constant containing the help identifier. This may be either a Help keyword or “context string.”                                       |
| <b>PRIMARY</b>     | Specifies at least one file must be placed in the procedure’s File Schematic.                                                                  |
| <i>message</i>     | A string constant containing a message that appears in the File Schematic next to the procedure’s Primary file.                                |
| <i>flag</i>        | If present, contains OPTIONAL (the file is not required), OPTKEY (the key is not required), or NOKEY (the file is not required to have a key). |
| <b>QUICK</b>       | Specifies the procedure has a wizard #UTILITY that runs when the <b>Use Procedure Wizard</b> box is checked.                                   |
| <i>wizard</i>      | The identifier (including template class, if necessary) of the wizard #UTILITY template.                                                       |

The **#PROCEDURE** statement begins a Procedure template. A Procedure template contains the Template and *target* language statements used to generate the source code for a procedure within your application. A #PROCEDURE section is terminated by the first occurrence of a Template Code Section (#PROGRAM, #MODULE, #PROCEDURE, #CONTROL, #CODE, #EXTENSION, or #GROUP) statement, or the end of the file. Within a Template set you may have multiple #PROCEDURE sections, but they must all have unique *name* parameters.

Example:

```
#PROCEDURE(ProcName1,'This is a sample window procedure'),WINDOW
#PROCEDURE(ProcName2,'This is a sample report procedure'),REPORT
#PROCEDURE(ProcName3,'This is a sample anything procedure'),WINDOW,REPORT
#PROCEDURE(Browse,'List with Wizard'),WINDOW,QUICK(BrowseWizard(Wizards))
```

## #GROUP (reusable statement group)

**#GROUP**( *symbol* [, [ *type* ] *parameters* [= *default* ] ] ) [, **AUTO** ] [, **PRESERVE** ] [, **HLP**( *helpid* ) ]

|                   |                                                                                                                                                                                                                                                                                                                                                                                                    |
|-------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>#GROUP</b>     | Begins a section of template code that may be inserted into another portion of the template.                                                                                                                                                                                                                                                                                                       |
| <i>symbol</i>     | A user-defined symbol used as the #GROUP's identifier.                                                                                                                                                                                                                                                                                                                                             |
| <i>type</i>       | The data type of a passed <i>parameter</i> : LONG, REAL, STRING, or * (asterisk). An asterisk (*) indicates it is a variable-parameter (passed by address), whose value may be changed by the #GROUP. LONG, REAL, and STRING indicates it is a value-parameter (passed by value), whose value is not changed by the #GROUP. If <i>type</i> is omitted, the <i>parameter</i> is passed as a STRING. |
| <i>parameters</i> | User-defined symbols by which values passed to the #GROUP are referenced. You may pass multiple <i>parameters</i> , each separated by commas, to a #GROUP. All specified <i>parameters</i> must be passed to the #GROUP; they may be omitted only if a <i>default</i> is supplied.                                                                                                                 |
| = <i>default</i>  | The value passed for an omitted <i>parameter</i> .                                                                                                                                                                                                                                                                                                                                                 |
| <b>AUTO</b>       | Opens a new scope for the group. This means that any #DECLARE statements in the #GROUP would not be available to the #PROCEDURE being generated. Passing <i>parameters</i> to a #GROUP implicitly opens a new scope.                                                                                                                                                                               |
| <b>PRESERVE</b>   | Preserves the current fixed instances of all built-in multi-valued symbols when the #GROUP is called and restores all those instances when the #GROUP code terminates.                                                                                                                                                                                                                             |
| <b>HLP</b>        | Specifies on-line help is available.                                                                                                                                                                                                                                                                                                                                                               |
| <i>helpid</i>     | A string constant containing the identifier to access the Help system. This may be either a Help keyword or "context string."                                                                                                                                                                                                                                                                      |

**#GROUP** defines the beginning of a section of code which is generated into the source. A #GROUP section may contain Template and/or target language code. The #GROUP section is terminated by the first occurrence of a Template Code Section (#PROGRAM, #MODULE, #PROCEDURE, #CONTROL, #CODE, #EXTENSION, or #GROUP) statement, or the end of the file. Within a single Template, separate #GROUP sections may not be defined with the same *symbol*. The *parameters* passed to a #GROUP fall into two categories: **value-parameters** and **variable-parameters**.

**Value-parameters** are declared as user-defined symbols, with an optional *type* and are "passed by value" (a copy of the value is passed) Either symbols or expressions may be passed as value-parameters. When a multi-valued symbol is passed as a value-parameter, only the current instance is passed.

**Variable-parameters** are declared as user-defined symbols with a prepended asterisk (\*) (and no *type*). A variable-parameter is “passed by address” and any change to its value by the #GROUP code changes the value of the passed symbol. Only symbols may be passed to a #GROUP as variable-parameters. When a multi-valued symbol is passed as a variable-parameter, all instances are passed.

The statements contained in the #GROUP section are generated by the #INSERT or #CALL statements. A #GROUP may contain #EMBED statements to define embedded source code points. A #GROUP may contain #PROMPT statements to obtain programmer input.

A #GROUP may act as a function if the #RETURN statement which passes control back from the #GROUP has a parameter. The value is returned to the CALL built-in procedure or directly to the expression in which the #GROUP is called as a function. If the #GROUP is called without the CALL built-in procedure and takes no parameters, open and close parentheses must be appended to the #GROUP symbol. For example, you may either place CALL(%MyGroup) in an expression or just %MyGroup().

Example:

```
#GROUP(%GenerateFormulas) #!A #GROUP without parameters
 #FOR(%Formula)
 #IF(%FormulaComputation)
%Formula = %FormulaComputation
 #ELSE
IF(%FormulaCondition)
 %Formula = %FormulaTrue
ELSE
 %Formula = %FormulaFalse
END
 #ENDIF
 #ENDFOR
#GROUP(%ChangeProperty,%MyField,%Property,%Value)
 #!A #GROUP that receives parameters
%MyField{%Property} = '%Value' #<!Change the %Property of %MyField

#GROUP(%SomeGroup, * %VarParm, LONG %ValParm)
 #!A #GROUP that receives a variable-parameter and a value-parameter
```

See Also:                   **#INSERT, #RETURN, CALL**

## #UTILITY (utility execution section)

```
#UTILITY(name, description) [, HLP(helpid)] [, WIZARD(procedure)]
```

|                    |                                                                                                                               |
|--------------------|-------------------------------------------------------------------------------------------------------------------------------|
| <b>#UTILITY</b>    | Begins a utility generation control.                                                                                          |
| <i>name</i>        | The name of the #UTILITY which identifies it for the Template Registry. This must be a valid Clarion label.                   |
| <i>description</i> | A string constant describing the utility section.                                                                             |
| <b>HLP</b>         | Specifies on-line help is available.                                                                                          |
| <i>helpid</i>      | A string constant containing the identifier to access the Help system. This may be either a Help keyword or "context string." |
| <b>WIZARD</b>      | Specifies the #UTILITY is used as a Wizard to generate a procedure or a complete application.                                 |
| <i>procedure</i>   | A string constant containing the fully qualified name of the #PROCEDURE for which it is a WIZARD.                             |

The #UTILITY statement marks the beginning of a utility execution control section. The section is terminated by the next Template Code Section (#PROGRAM, #MODULE, #PROCEDURE, #CONTROL, #CODE, #EXTENSION, #UTILITY, or #GROUP) statement. The Template statements contained in this section control the utility execution process. Multiple #UTILITY sections are allowed in a single Template set.

The #UTILITY section is very similar to the #APPLICATION section, in that it allows you to produce output from the application. The purpose of #UTILITY is to provide extensible supplemental utilities for such things as program documentation, or a tree diagram of procedure calls. The list of registered utilities appears in the Utilities menu in the Clarion for Windows environment. #UTILITY with the WIZARD attribute specifies it contains a #SHEET with #TABs that display one tab at a time, guiding the user through the prompts.

Example:

```
#UTILITY(ProcCallTree, 'Output procedure call tree')
 #CREATE(%Application & '.TRE')
 Procedure Call Tree: for %Application
 #INSERT(%DisplayTree, %FirstProcedure, '|', '|')
 #CLOSE
 #!*****
 #GROUP(%DisplayTree, %ThisProc, %Level, %NextIndent)
 #FIX(%Procedure, %ThisProc)
 %Level+-%ThisProc (%ProcedureTemplate)
 #FOR(%ProcedureCalled)
 #IF(INSTANCE(%ProcedureCalled) = ITEMS(%ProcedureCalled))
 #INSERT(%DisplayTree, %ProcedureCalled, %Level & %NextIndent, '|')
 #ELSE
 #INSERT(%DisplayTree, %ProcedureCalled, %Level & %NextIndent, '|')
 #ENDIF
 #ENDFOR
```

## #CODE (define a code template)

```
#CODE(name,description [,target])[, SINGLE][, HLP(helpid)][, PRIMARY(message [, flag])
[, DESCRIPTION(expression)][, ROUTINE][, PRIORITY(number)]
[, REQ(addition [, | BEFORE |])][, | FIRST |]
| AFTER | | LAST |]
```

|                    |                                                                                                                                                                                              |
|--------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>#CODE</b>       | Begins a code template that generates source into an embedded source code point.                                                                                                             |
| <i>name</i>        | The label of the code template. This must be a valid Clarion label.                                                                                                                          |
| <i>description</i> | A string constant describing the code template.                                                                                                                                              |
| <i>target</i>      | A string constant that specifies the source language the code template generates. If omitted, it defaults to Clarion. This restricts the #CODE to matching <i>target</i> language use, only. |
| <b>SINGLE</b>      | Specifies the #CODE may be used only once in a given procedure (or program, if the embedded source code point is global).                                                                    |
| <b>HLP</b>         | Specifies on-line help is available.                                                                                                                                                         |
| <i>helpid</i>      | A string constant containing the identifier to access the Help system. This may be either a Help keyword or “context string.”                                                                |
| <b>PRIMARY</b>     | Specifies a primary file for the code template must be placed in the procedure’s File Schematic.                                                                                             |
| <i>message</i>     | A string constant containing a message that appears in the File Schematic next to the #CODE’s Primary file.                                                                                  |
| <i>flag</i>        | Either OPTIONAL (the file is not required), OPTKEY (the key is not required), or NOKEY (the file is not required to have a key).                                                             |
| <b>DESCRIPTION</b> | Specifies the display description of a #CODE that may be used multiple times in a given application or procedure.                                                                            |
| <i>expression</i>  | A string constant or expression that contains the description to display.                                                                                                                    |
| <b>ROUTINE</b>     | Specifies the generated code is not automatically indented from column one.                                                                                                                  |
| <b>PRIORITY</b>    | Specifies the order in which the #CODE is generated into the embed point. The lowest value generates first.                                                                                  |
| <i>number</i>      | An integer constant in the range 1 to 10000.                                                                                                                                                 |

|                 |                                                                                                                                                                                                     |
|-----------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>REQ</b>      | Specifies the #CODE requires a previously placed #CODE, #CONTROL, or #EXTENSION before it may be used. It also means all prompts and variables of the required <i>addition</i> are available to it. |
| <i>addition</i> | The name of the previously placed #CODE, #CONTROL, or #EXTENSION template, from any template set.                                                                                                   |
| <b>BEFORE</b>   | Legacy attribute, replaced by PRIORITY.                                                                                                                                                             |
| <b>AFTER</b>    | Legacy attribute, replaced by PRIORITY.                                                                                                                                                             |
| <b>FIRST</b>    | Equivalent to PRIORITY(1).                                                                                                                                                                          |
| <b>LAST</b>     | Equivalent to PRIORITY(10000).                                                                                                                                                                      |

**#CODE** defines the beginning of a code template which can generate code into embedded source code points. A #CODE section may contain Template and/or target language code. The #CODE section is terminated by the first occurrence of a Template Code Section (#PROGRAM, #MODULE, #PROCEDURE, #CONTROL, #CODE, #EXTENSION, or #GROUP) statement, or the end of the file. Within a single Template set, separate #CODE sections may not be defined with the same *name*.

#CODE generates its code into a #EMBED embedded source code point. The generated code is automatically indented when placed in ROUTINES, unless the ROUTINE attribute is present. A #CODE section may contain #PROMPT statements to prompt for the values needed to generate proper source code. It may also contain #EMBED statements, which become active only if the #CODE section is used.

You can use the #WHERE statement to limit the availability of the #CODE to those embedded source code points where the generated code would be appropriate. A #CODE may contain multiple #WHERE statements to explicitly define all the valid embedded source code points in which it may appear. #RESTRICT can also further restrict the availability of the #CODE based on an expression or Template language statements.

The #AT/#ENDAT structure allows a single #CODE to generate code into multiple embedded source code points to support its functionality.

Example:

```
#CODE(ChangeProperty,'Change control property')
 #WHERE(%SetupWindow.%ProcedureRoutines) #!Appropriate only after window open
 #PROMPT('Control to change',CONTROL),%MyField,REQ
 #PROMPT('Property to change',@S20),%Property,REQ
 #PROMPT('New Value',@S20),%Value,REQ
 %MyField{%Property} = '%Value' #<!Change the %Property of %MyField
```

See Also:           **#EMBED, #WHERE, #RESTRICT, #AT**

## #CONTROL (define a control template)

```
#CONTROL(name, description) [, MULTI] [, PRIMARY(message [, flag])] [, SHOW]
 [, WINDOW] [, REPORT] [, WRAP(control)] [, PRIORITY(number)]
 [, REQ(addition [, | BEFORE |])] [, | FIRST |] [, DESCRIPTION(expression)]]
 | AFTER | | LAST |
CONTROLS [, REPORT]
 control statements [, #REQ]
END
```

|                    |                                                                                                                                                                                         |
|--------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>#CONTROL</b>    | Begins a code template that generates a set of controls into a window and the source code required to manipulate them into embedded source code points.                                 |
| <i>name</i>        | The label of the template (must be a valid Clarion label).                                                                                                                              |
| <i>description</i> | A string constant describing the control template.                                                                                                                                      |
| <b>MULTI</b>       | Specifies the #CONTROL may be used multiple times in a given window.                                                                                                                    |
| <b>PRIMARY</b>     | Specifies a primary file for the set of controls must be placed in the procedure's File Schematic.                                                                                      |
| <i>message</i>     | A string constant containing a message that appears in the File Schematic next to the #CONTROL's Primary file.                                                                          |
| <i>flag</i>        | Either OPTIONAL (the file is not required), OPTKEY (the key is not required), or NOKEY (the file is not required to have a key).                                                        |
| <b>SHOW</b>        | Specifies the #CONTROL prompts are placed on the procedure properties window.                                                                                                           |
| <b>WINDOW</b>      | Tells the Application Generator to make the #CONTROL available in the Window Formatter. This is the default setting if both WINDOW and REPORT are omitted.                              |
| <b>REPORT</b>      | Tells the Application Generator to make the #CONTROL available in the Report Formatter. If omitted, the #CONTROL may not be placed in a REPORT.                                         |
| <b>WRAP</b>        | Specifies the #CONTROL template is offered as an option for the <i>control</i> when the "Translate controls to control templates when populating" option is set in Application Options. |
| <i>control</i>     | The data type of the control for which the #CONTROL is a viable alternative.                                                                                                            |
| <b>PRIORITY</b>    | Specifies the order in which the #CONTROL is generated. The lowest value generates first.                                                                                               |
| <i>number</i>      | An integer constant in the range 1 to 10000.                                                                                                                                            |

|                    |                                                                                                                                                                                                                                                                                         |
|--------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>REQ</b>         | Specifies the #CONTROL requires a previously placed #CODE, #CONTROL, or #EXTENSION before it may be used.                                                                                                                                                                               |
| <i>addition</i>    | The name of the previously placed #CODE, #CONTROL, or #EXTENSION.                                                                                                                                                                                                                       |
| <b>BEFORE</b>      | Legacy attribute, replaced by PRIORITY.                                                                                                                                                                                                                                                 |
| <b>AFTER</b>       | Legacy attribute, replaced by PRIORITY.                                                                                                                                                                                                                                                 |
| <b>FIRST</b>       | Equivalent to PRIORITY(1).                                                                                                                                                                                                                                                              |
| <b>LAST</b>        | Equivalent to PRIORITY(10000).                                                                                                                                                                                                                                                          |
| <b>DESCRIPTION</b> | Specifies the display description of a #CONTROL that may be used multiple times in a given application or procedure.                                                                                                                                                                    |
| <i>expression</i>  | A string constant or expression that contains the description to display.                                                                                                                                                                                                               |
| <b>CONTROLS</b>    | Specifies the <i>controls</i> for the #CONTROL, and must be terminated with an END statement. This is a “pseudo-Clarion keyword” in that, if you replace the CONTROLS statement with a WINDOW statement, you can use the Text Editor’s Window Formatter to create the <i>controls</i> . |
| <b>REPORT</b>      | Specifies the <i>controls</i> for the #CONTROL will be placed in a REPORT structure. This allows report-specific control attributes (like PAGENO) to be used. It also indicates the AT attribute measurement unit is THOUS.                                                             |
| <i>controls</i>    | Control declarations that specify the control set belonging to the #CONTROL.                                                                                                                                                                                                            |
| <b>#REQ</b>        | Specifies the <i>control</i> is required. If deleted from the window or report, the entire #CONTROL (including all its <i>controls</i> ) is deleted.                                                                                                                                    |

**#CONTROL** defines the beginning of a code template containing a “matched set” of controls to populate into a window or report as a group. It also generates the source code required for their correct operation into embedded source code points. A #CONTROL section may contain Template and/or target language code. The #CONTROL section is terminated by the first occurrence of a Template Code Section (#PROGRAM, #MODULE, #PROCEDURE, #CONTROL, #CODE, #EXTENSION, or #GROUP) statement, or the end of the file. Within a single Template set, separate #CONTROL sections may not be defined with the same *name*.

#CONTROL generates the code to operate its *controls* into #EMBED embedded source code points using the #AT/#ENDAT structure. #RESTRICT can restrict use of the #CONTROL based on an expression or Template language statements.

A `#CONTROL` section may contain `#PROMPT` statements to prompt for the values needed to generate proper source code. These prompts appear on the Actions window in the environment. It may also contain `#EMBED` statements which become active only if the `#CONTROL` section is used.

The  $x$  and  $y$  parameters of the `AT` attribute of the *controls* in the `#CONTROL` set determine the positioning of the *control* relative to the last control in the `#CONTROL` set placed on screen (or relative to the window, if first). If these parameters are omitted, the programmer is prompted for the position to place the *control*. This makes it simple to populate an entire set of *controls* without requiring the programmer to place each one individually.

The `WRAP` attribute specifies the `#CONTROL` is offered as an option to the programmer when the “Translate controls to control templates when populating” option is set in Application Options. The control parameter specifies the type of control to which the `#CONTROL` applies. This makes the `#CONTROL` a “wrapper” for the control type, such that, when the programmer populates the control in the formatter, a dialog appears offering the choice of populating either the control itself, or the `#CONTROL` template. For example, with the `WRAP(LIST)` attribute on a `#CONTROL`, when the programmer attempts to populate a `LIST` control a dialog appears offering the opportunity to use either the `#CONTROL` template (which generates executable code to “drive” the control) or the `LIST` control itself (requiring the programmer to write the “driving” code for the control).

Example:

```
#CONTROL(BrowseList,'Add Browse List controls')
#PROMPT('Allow Inserts',CHECK),%InsertAllowed,DEFAULT(1)
#ENABLE(%InsertAllowed)
#PROMPT('Insert Hot Key',@s20),%InsertHotKey,DEFAULT('InsertKey')
#ENDENABLE
#PROMPT('Allow Changes',CHECK),%ChangeAllowed,DEFAULT(1)
#ENABLE(%ChangeAllowed)
#PROMPT('Change Hot Key',@s20),%ChangeHotKey,DEFAULT('CtrlEnter')
#ENDENABLE
#PROMPT('Allow Deletes',CHECK),%DeleteAllowed,DEFAULT(1)
#ENABLE(%DeleteAllowed)
#PROMPT('Delete Hot Key',@s20),%DeleteHotKey,DEFAULT('DeleteKey')
#ENDENABLE
#PROMPT('Update Procedure',PROCEDURE),%UpdateProc
CONTROLS
LIST,AT(, ,270,99),USE(?List),IMM,FROM(Queue:Browse),#REQ
BUTTON('Insert'),AT(, ,40,15),USE(?Insert),MSG('Add record')
BUTTON('Change'),AT(, ,40,15),USE(?Change),DEFAULT,MSG('Change Record')
BUTTON('Delete'),AT(, ,40,15),USE(?Delete),MSG('Delete record')
END
#!
#AT(BeforeAccept)
#IF(%InsertAllowed)
?Insert{PROP:Key} = %InsertHotKey
#ENDIF
#IF(%ChangeAllowed)
?Change{PROP:Key} = %ChangeHotKey
#ENDIF
#IF(%DeleteAllowed)
?Delete{PROP:Key} = %DeleteHotKey
#ENDIF
#ENDAT
#!
#AT(%ControlEvent),WHERE(%ControlOriginal='?Insert' AND %ControlEvent='Accepted')
#IF(%InsertAllowed)
Action = AddRecord
%UpdateProc
#ENDIF
#ENDAT
#!
#AT(%ControlEvent),WHERE(%ControlOriginal='?Chg' AND %ControlEvent='Accepted')
#IF(%ChangeAllowed)
Action = ChangeRecord
%UpdateProc
#ENDIF
#ENDAT
#!
#AT(%ControlEvent),WHERE(%ControlOriginal='?Delete' AND %ControlEvent='Accepted')
#IF(%DeleteAllowed)
Action = DeleteRecord
%UpdateProc
#ENDIF
#ENDAT
```

See Also:

**#EMBED, #WHERE, #RESTRICT, #AT**

## #EXTENSION (define an extension template)

```
#EXTENSION(name, description [, target]) [, MULTI] [, DESCRIPTION(expression)]]
 [, SHOW] [, PRIMARY(message [, flag])] [, | APPLICATION([child(chain)])]]
 | PROCEDURE |
 [, REQ(addition [, | BEFORE |]) [, | FIRST |] [, PRIORITY(number)]
 | AFTER | | LAST |
```

|                     |                                                                                                                                                                                 |
|---------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>#EXTENSION</b>   | Begins an extension template that generates code into embedded source code points to add some functionality not associated with specific controls.                              |
| <i>name</i>         | The label of the extension template. This must be a valid Clarion label.                                                                                                        |
| <i>description</i>  | A string constant describing the extension template.                                                                                                                            |
| <i>target</i>       | A string constant that specifies the source language the extension template generates. If omitted, it defaults to Clarion.                                                      |
| <b>MULTI</b>        | Specifies the #EXTENSION may be used multiple times in a given application or procedure.                                                                                        |
| <b>DESCRIPTION</b>  | Specifies the display description of a #EXTENSION that may be used multiple times in a given application or procedure.                                                          |
| <i>expression</i>   | A string constant or expression that contains the description to display.                                                                                                       |
| <b>SHOW</b>         | Specifies the #EXTENSION prompts are placed on the procedure properties window.                                                                                                 |
| <b>PRIMARY</b>      | Specifies a primary file for the extension must be placed in the procedure's File Schematic.                                                                                    |
| <i>message</i>      | A string constant containing a message that appears in the File Schematic next to the #EXTENSION's Primary file.                                                                |
| <i>flag</i>         | Either OPTIONAL (the file is not required), OPTKEY (the key is not required), or NOKEY (the file is not required to have a key).                                                |
| <b>APPLICATION</b>  | Tells the Application Generator to make the #EXTENSION available only at the global level.                                                                                      |
| <i>child(chain)</i> | The name of a #EXTENSION with the PROCEDURE attribute to automatically populate into every generated procedure when the #EXTENSION with the APPLICATION attribute is populated. |
| <b>PROCEDURE</b>    | Tells the Application Generator to make the #EXTENSION available only at the local level.                                                                                       |

|                 |                                                                                                             |
|-----------------|-------------------------------------------------------------------------------------------------------------|
| <b>REQ</b>      | Specifies the #EXTENSION requires a previously placed #CODE, #CONTROL, or #EXTENSION before it may be used. |
| <i>addition</i> | The name of the previously placed #CODE, #CONTROL, or #EXTENSION.                                           |
| <b>BEFORE</b>   | Legacy attribute, replaced by PRIORITY.                                                                     |
| <b>AFTER</b>    | Legacy attribute, replaced by PRIORITY.                                                                     |
| <b>FIRST</b>    | Equivalent to PRIORITY(1).                                                                                  |
| <b>LAST</b>     | Equivalent to PRIORITY(10000).                                                                              |
| <b>PRIORITY</b> | Specifies the order in which the #EXTENSION is generated. The lowest value generates first.                 |
| <i>number</i>   | An integer constant in the range 1 to 10000.                                                                |

**#EXTENSION** defines the beginning of an extension template containing code to generate into the application or procedure to provide some functionality not directly associated with any control. A #EXTENSION section may contain Template and/or target language code. The #EXTENSION section is terminated by the first occurrence of a Template Code Section (#PROGRAM, #MODULE, #PROCEDURE, #CONTROL, #CODE, #EXTENSION, or #GROUP) statement, or the end of the file. Within a single Template set, separate #EXTENSION sections may not be defined with the same *name*.

#EXTENSION can only generate code into #EMBED embedded source code points using the #AT/#ENDAT structure. A #EXTENSION section may contain #PROMPT statements to prompt for the values needed to generate proper source code. These prompts appear when you edit an Extension from the Extensions button in the environment. It may also contain #EMBED statements which become active only if the #EXTENSION section is used.

#RESTRICT can restrict appearance of the #EXTENSION in the list of available extensions based on an expression or Template language statements.

Example:

```
#EXTENSION(Security,'Add password'),PROCEDURE
 #PROMPT('Password File',FILE),%PasswordFile,REQ
 #PROMPT('Password Key',KEY(%PasswordFile)),%PasswordFileKey,REQ
 #PROMPT('Password Field',COMPONENT(%PasswordFileKey)),%PasswordFileKeyField,REQ
 #AT(%DataSectionBeforeWindow)
LocalPswd STRING(10)
SecurityWin WINDOW
 ENTRY(@s10),USE(LocalPswd),REQ,PASSWORD
 BUTTON('Cancel'),KEY(EscKey),USE(?CancelPswd)
 END
 #ENDAT
 #AT(%ProcedureSetup)
OPEN(SecurityWin)
ACCEPT
CASE ACCEPTED()
OF ?LocalPswd
 %PasswordFileKeyField = LocalPswd
 GET(%PasswordFile,%PasswordFileKey)
 IF NOT ERRORCODE()
 LocalPswd = 'OK'
 END
 BREAK
OF ?CancelPswd
 CLEAR(LocalPswd)
 BREAK
END
END
CLOSE(SecurityWin)
IF LocalPswd <> 'OK' THEN RETURN.
#ENDAT
```

See Also:           **#EMBED, #WHERE, #RESTRICT, #AT**

## Embed Points

### #EMBED (define embedded source point)

```
#EMBED(identifier [, descriptor]) [, symbol] [, HLP(helpid)] [, DATA] [, HIDE]
 [, WHERE(expression)] [, MAP(mapsymbol, description)] [, LABEL] [, NOINDENT]
 [, DEPRECATED] [, PREPARE(parameters)] [, TREE(displaytext)]
```

|                    |                                                                                                                                                                                                                                                                                      |
|--------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>#EMBED</b>      | Identifies an explicit position in the Template where the programmer may place their own source code.                                                                                                                                                                                |
| <i>identifier</i>  | A user-defined template symbol which identifies the embedded source code point for the Application Generator.                                                                                                                                                                        |
| <i>descriptor</i>  | A string constant containing a description of the embedded source code's position in the Template. This is the string displayed in the list of available embedded source code windows for a procedure Template (normally omitted if the HIDE attribute is present).                  |
| <i>symbol</i>      | A multi-valued template symbol or a literal string. You may have multiple <i>symbols</i> on a single #EMBED statement. This may also have a description to specify the text to display in the embedded source tree appended to it with square brackets (i.e. %symbol[%description]). |
| <b>HLP</b>         | Specifies on-line help is available for the #EMBED.                                                                                                                                                                                                                                  |
| <i>helpid</i>      | A string constant containing the identifier to access the Help system. This may be either a Help keyword or "context string."                                                                                                                                                        |
| <b>DATA</b>        | Specifies the embed point is in a data section, so the Text Editor's Window and Report Formatters can be used.                                                                                                                                                                       |
| <b>HIDE</b>        | Specifies the source code point does not appear in the tree of available embedded source code points. Therefore, the #EMBED is only available for #CODE, #CONTROL, or #EXTENSION code generation.                                                                                    |
| <b>WHERE</b>       | Specifies the #EMBED is available only for those instances of the <i>symbol</i> where the <i>expression</i> is true.                                                                                                                                                                 |
| <i>expression</i>  | An expression that specifies the condition.                                                                                                                                                                                                                                          |
| <b>MAP</b>         | Maps the <i>description</i> to the <i>symbol</i> for display in the embedded source tree. You may have as many MAP attributes as there are <i>symbols</i> .                                                                                                                          |
| <i>mapsymbol</i>   | Names which of the #EMBED <i>symbols</i> the MAP references.                                                                                                                                                                                                                         |
| <i>description</i> | An expression that specifies the text to display in the embedded source tree.                                                                                                                                                                                                        |

|                    |                                                                                                                                                                                                                                                                                                                                                                          |
|--------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>LABEL</b>       | Specifies the embed point is generated in column one (1) so the Text Editor will correctly color-code the labels.                                                                                                                                                                                                                                                        |
| <b>NOINDENT</b>    | Specifies the code generated into the embed point retains the indentation from its source (either generated or entered in by the programmer)—source in column 1 is generated into column 1, no matter where the #EMBED is placed.                                                                                                                                        |
| <b>DEPRECATED</b>  | Specifies the #EMBED is displayed in the list of available embed points only if there is already code in it. This indicates an embed point which has been superceded and is only present to allow the existing code in it to be moved to a new embed or deleted.                                                                                                         |
| <b>PREPARE</b>     | Executes preparatory code before evaluating the #EMBED <i>symbol s</i> . The <i>n</i> <sup>th</sup> <i>parameter</i> to PREPARE is evaluated immediately before the <i>n</i> <sup>th</sup> <i>symbol</i> is evaluated.                                                                                                                                                   |
| <i>parameters</i>  | A comma-delimited list of expressions or calls to #GROUPs. You may have as many PREPARE <i>parameters</i> as there are <i>symbols</i> on the #EMBED.                                                                                                                                                                                                                     |
| <b>TREE</b>        | Specifies what will be added to the embeds dialog tree. This overrides the MAP attribute.                                                                                                                                                                                                                                                                                |
| <i>displaytext</i> | A comma-delimited list of strings (constants or variables) containing tree level text, color, and priority information. Each tree level may be a separate entry or contained in a single separated by vertical bar ( ) character. Following the text to display you may have optional color and priority information contained in curly braces, in the following format: |

```
'TextToDisplay{{Color(0FFh),priority(5000}}'
```

The color information may be either a number or the text for a color EQUATE from the EQUATES.CLW file.

**#EMBED** identifies an explicit position in the Template where the programmer may call a procedure, generate code from a code template, or place their own custom embedded source code within the procedure. The Application Generator prompts the programmer for the procedure to call, or the code template to use, or calls the Text Editor to allow the programmer to write the embedded source code. #EMBED is also used as the destination of all the source automatically generated by #CODE, #CONTROL, and #EXTENSION template sections. If no code is written in the embedded source code point by the programmer or any code template, control template, or extension template, no code is generated.

In a #PROCEDURE section, the source code is automatically placed in the exact column position at which #EMBED is located within the Template. If #EMBED is directly placed in the data section of a #PROGRAM, #MODULE, or #PROCEDURE, it must be in column one (1) of the Template file (so the embedded code may contain data labels). If the

#EMBED statement has the DATA attribute, the Window and Report Formatters in the Text Editor are available for use. In executable code sections, #EMBED may be placed in column one, but that is not required.

#EMBED is valid in a #GROUP section, however, this should be used with care. Since it is possible for a #GROUP to be recursive (call itself), it is possible to create embedded source code that is repeated within each iteration of the recursive #GROUP's generated code. The source code is generated in the same relative column position as the code generated from the #GROUP. A #EMBED using the *symbol* attribute is used within a #FOR statement to allow a different piece of embedded source to be inserted for each instance of the *symbol*. It can also be used within #FOR, #LOOP, and/or recursive #GROUPs for the current instance of the symbol (if it has been #FIXed). The MAP attribute allows you to replace a *description* for the *symbol* in the embedded source tree.

Example:

```
#PROCEDURE(SampleProc,'This is a sample procedure'),WINDOW
#EMBED(%DataSection,'Data Section Source Code Window'),DATA #!Source code in column 1
CODE !Begin executable code
#EMBED(%SetupProc,'Code Section Source Code Window 1') #!Source code in column 3
OPEN(Screen) !Open window
ACCEPT !Event handler
CASE SELECTED() !Handle field-selection events
#FOR(%Control)
OF %Control
 #EMBED(%ScreenFieldSetupEmbed,'Field Selected Embed'),%Control
#ENDFOR
END
CASE ACCEPTED() !Handle field-action events
#FOR(%Control)
OF %Control
 #EMBED(%ScreenFieldEditEmbed,'Field Accepted Embed'),%Control
#ENDFOR
.
.#EMBED(%CustomRoutines,'Code Section Source Code Window 2'),LABEL #!Source in col 1
```

## #AT (insert code in an embed point)

```
#AT(location [, instances]) [, WHERE(expression)] [, AUTO] [, PRESERVE]
 [, PRIORITY(number)] [, DESCRIPTION(text)] [, FIRST] [, LAST]
 statements
#ENDAT
```

|                    |                                                                                                                                                                                                                                                                                     |
|--------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>#AT</b>         | Specifies a <i>location</i> to generate <i>statements</i> .                                                                                                                                                                                                                         |
| <i>location</i>    | An #EMBED <i>identifier</i> . This may be a #EMBED for a procedure that comes from another template set.                                                                                                                                                                            |
| <i>instances</i>   | The <i>location</i> parameters that identify the embedded source code point for a multi-valued #EMBED <i>identifier</i> . There may as many <i>instance</i> parameters as are required to explicitly identify the embedded source code point. These <i>instances</i> are omissible. |
| <b>WHERE</b>       | More closely specifies the #AT <i>location</i> as only those embed points where the <i>expression</i> is true.                                                                                                                                                                      |
| <i>expression</i>  | An expression that specifies exact placement.                                                                                                                                                                                                                                       |
| <b>AUTO</b>        | Opens a new scope for the #AT. This means that any #DECLARE statements in the #AT would not be available to the #PROCEDURE being generated.                                                                                                                                         |
| <b>PRESERVE</b>    | Preserves the current fixed instances of all built-in multi-valued symbols when the #AT is called and restores all those instances when the #AT code terminates.                                                                                                                    |
| <b>PRIORITY</b>    | Specifies the order in which the #AT is generated into the <i>location</i> . The lowest value generates first.                                                                                                                                                                      |
| <i>number</i>      | An integer constant in the range 1 to 10000.                                                                                                                                                                                                                                        |
| <b>DESCRIPTION</b> | Specifies the <i>text</i> generated to describe the PRIORITY's actual embed point in the embed tree and Embeditor (as a comment).                                                                                                                                                   |
| <i>text</i>        | A string to describe the PRIORITY's actual embed point in the generated source.                                                                                                                                                                                                     |
| <b>FIRST</b>       | Equivalent to PRIORITY(1).                                                                                                                                                                                                                                                          |
| <b>LAST</b>        | Equivalent to PRIORITY(10000).                                                                                                                                                                                                                                                      |
| <i>statements</i>  | Template and/or target language code.                                                                                                                                                                                                                                               |
| <b>#ENDAT</b>      | Terminates the section.                                                                                                                                                                                                                                                             |

The #AT structure specifies a *location* to generate *statements*. #AT is valid only in a #CONTROL, #CODE, or #EXTENSION templates, and is used to allow them to generate *statements* into multiple *locations*. The #AT structure must terminate with #ENDAT.

The WHERE clause allows you to create an *expression* that can specify a single specific instance of a #EMBED that has a *symbol* attribute. You may not place #AT within any type of conditional structure (such as #IF or #CASE). If you need to conditionally generate the code, place the #IF or #CASE structure within the #AT structure.

Example:

```
#CONTROL(BrowseList,'Add Browse List controls')
 #AT(%ControlEvent,'?Insert','Accepted'),PRIORITY(5000)
 #IF(%InsertAllowed)
GlobalRequest = InsertRecord
%UpdateProc
 #ENDIF
 #ENDAT
#!
```

See Also:

**#EMBED, #CODE, #CONTROL, #EXTENSION, #RESTRICT**

## #PRIORITY (set new embed priority level)

**#PRIORITY( *number* ) [ , DESCRIPTION( *text* ) ]**

|                    |                                                                                                                                    |
|--------------------|------------------------------------------------------------------------------------------------------------------------------------|
| <b>#PRIORITY</b>   | Specifies a new priority level for the code following the #PRIORITY statement.                                                     |
| <i>number</i>      | An integer constant in the range 1 to 10000.                                                                                       |
| <b>DESCRIPTION</b> | Specifies the <i>text</i> generated to describe the #PRIORITY's actual embed point in the embed tree and Embeditor (as a comment). |
| <i>text</i>        | A string to describe the #PRIORITY's actual embed point in the generated source.                                                   |

The **#PRIORITY** statement specifies a new priority *number* for the code statements following the #PRIORITY.

Example:

```
#CONTROL(BrowseList,'Add Browse List controls')
 #AT(%ControlEvent,'?Insert','Accepted'),PRIORITY(5000)#!Start at priority 5000
GlobalRequest = InsertRecord #!Goes into priority 5000
 #PRIORITY(8500)
%UpdateProc #!Goes into priority 8500
#ENDAT
#!
```

See Also:           **#AT, #EMBED, #CODE, #CONTROL, #EXTENSION**

## #ATSTART (template initialization code)

```
#ATSTART
 statements
#ENDAT
```

**#ATSTART** Specifies template code to execute before the #PROCEDURE, #CODE, #CONTROL, or #EXTENSION generates.

*statements* Template language code.

**#ENDAT** Terminates the section.

The **#ATSTART** structure specifies template code to execute before the #PROCEDURE, #CODE, #CONTROL, or #EXTENSION generates its code. It will also execute when the application is loaded into the Clarion environment and when a template is first populated into the application. Therefore, the *statements* should normally only contain Template language. #ATSTART is usually used to initialize internal template variables.

You may not place #ATSTART within any type of conditional structure (such as #IF or #CASE). If you need to conditionally generate the code, place the #IF or #CASE structure within the #ATSTART structure.

Example:

```
#CONTROL(BrowseList,'Add Browse List controls')
 #ATSTART
 #FIX(%Control,%ListBox)
 #DECLARE(%ListPre)
 #SET(%ListPre,'List' & %ActiveTemplateInstance & ':')
 #!Makes %ListPre contain "List#:"
 #ENDAT
 #AT(%DataSectionBeforeWindow)
%ListPre:Scroll LONG !Scroll for %Control
%ListPre:Chioce LONG !Choice for %Control
#ENDAT
```

See Also:           **#PROCEDURE, #CODE, #CONTROL, #EXTENSION**

## #ATEND (template reset code)

---

```
#ATEND
 statements
#ENDAT
```

**#ATEND** Specifies template code to execute after the #PROCEDURE, #CODE, #CONTROL, or #EXTENSION generates.

*statements* Template language code.

**#ENDAT** Terminates the section.

The **#ATEND** structure specifies template code to execute after the #PROCEDURE, #CODE, #CONTROL, or #EXTENSION generates its code. Therefore, the *statements* should only contain Template language. #ATEND is usually used to reset internal template variables. You may not place #ATEND within any type of conditional structure (such as #IF or #CASE). If you need to conditionally generate the code, place the #IF or #CASE structure within the #ATEND structure.

Example:

```
#CONTROL(BrowseList,'Add Browse List controls')
#ATEND
 #SET(%ListQueue,%NULL)
#ENDAT
```

See Also: **#PROCEDURE, #CODE, #CONTROL, #EXTENSION**

## #CONTEXT (set template code generation context)

```
#CONTEXT(section [, instance])
 statements
#ENDCONTEXT
```

|                    |                                                                                                                          |
|--------------------|--------------------------------------------------------------------------------------------------------------------------|
| <b>#CONTEXT</b>    | Sets the code generation context to emulate generating the named <i>section</i> .                                        |
| <i>section</i>     | One of the following valid symbols: %Application, %Procedure, %Module, or %Program.                                      |
| <i>instance</i>    | A symbol containing a valid instance number of a code, control ,or extension template used in the named <i>section</i> . |
| <i>statements</i>  | Template language code.                                                                                                  |
| <b>#ENDCONTEXT</b> | Terminates the context change.                                                                                           |

The **#CONTEXT** structure specifies template code that executes as if the source code for the named *section* were being generated. Once the context has been set, all the prompt symbols , declared symbols, and embeds in the named *section* are in scope and available for use in the **#CONTEXT** *statements*. If an *instance* is specified, the prompts for the corresponding component are brought into scope. **#CONTEXT** is valid for use anywhere in template code.

The key to understanding the use of the **#CONTEXT** structure is “**as if the source code for the named *section* were being generated.**” This statement means that the *statements* are evaluated as if **#GENERATE** were executing. For example, a **#EMBED** statement within a **#CONTEXT** structure does not name a new embed point but instead, generates the contents of the named embed point.

Example:

```
#CODE(StealSomeCode,'Preparing to Process the Window Code Stealer')
#PROMPT('Procedure Name',FROM(%Procedure)),%ProcToStealFrom,REQ
#FIX(%Procedure,%ProcToStealFrom)
#CONTEXT(%Procedure)
 #EMBED(%BeforeAccept,'Preparing to Process the Window')
#ENDCONTEXT
```

See Also: **#CODE**, **#CONTROL**, **#EXTENSION**

## #EMPTYEMBED (generate empty embed point comments)

**#EMPTYEMBED**( *text* [, *condition* ] )

**#EMPTYEMBED** Generates comments into empty embed points.

*text* A string constant or constant expression containing the text to place in the empty embed point.

*condition* An expression that, when true, allows the comments to generate.

The **#EMPTYEMBED** statement specifies that comments generate into all embed points in which the user has not entered code. This will not generate comments for embed points in which the user has entered code or in which the templates have generated code.

The output *condition* is usually the value of a global prompt.

The comment *text* may use the %EmbedID, %EmbedDescription, and %EmbedParameters built-in symbols to identify the embed point:

%EmbedID The current embed point's identifying symbol.

%EmbedDescription  
The current embed point's description.

%EmbedParameters  
The current embed point's current instance, as a comma-delimited list.

Example:

```
#! This example is complete extension template, ready to use
#EXTENSION(EmptyEmbeds,'Empty Embed Comments'),APPLICATION
#PROMPT('Generate Empty EMBED Comments',CHECK),%EmptyEmbeds
#EMPTYEMBED('!Embed: ' & %EmbedDescription & ' ' & %EmbedParameters,%EmptyEmbeds)
```

See Also: **#PREEMBED**, **#POSTEMBED**

## #POSTEMBED (generate ending embed point comments)

**#POSTEMBED**( *text* [, *condition* ] )

**#POSTEMBED** Generates comments at the end of embed point code.

*text* A string constant or constant expression containing the text to place in the embed point.

*condition* An expression that, when true, allows the comments to generate.

The **#POSTEMBED** statement specifies that comments generate at the end of embed points that contain code. The output *condition* is usually the value of a global prompt.

The comment *text* may use the %EmbedID, %EmbedDescription, and %EmbedParameters built-in symbols to identify the embed point:

%EmbedID The current embed point's identifying symbol.

%EmbedDescription The current embed point's description.

%EmbedParameters The current embed point's current instance, as a comma-delimited list.

Example:

```
#POSTEMBED('! After Embed Point: ' & %EmbedID & ' ' & %EmbedDescription & ' ' & |
 %EmbedParameters,%GenerateEmbedComments)
```

See Also: **#PREEMBED**, **#EMPTYEMBED**

## #PREEMBED (generate beginning embed point comments)

**#PREEMBED**( *text* [, *condition* ] )

|                  |                                                                                           |
|------------------|-------------------------------------------------------------------------------------------|
| <b>#PREEMBED</b> | Generates comments at the beginning of embed point code.                                  |
| <i>text</i>      | A string constant or constant expression containing the text to place in the embed point. |
| <i>condition</i> | An expression that, when true, allows the comments to generate.                           |

The **#PREEMBED** statement specifies that comments generate at the beginning of embed points that contain code. The output *condition* is usually the value of a global prompt.

The comment *text* may use the %EmbedID, %EmbedDescription, and %EmbedParameters built-in symbols to identify the embed point:

|                   |                                                                        |
|-------------------|------------------------------------------------------------------------|
| %EmbedID          | The current embed point's identifying symbol.                          |
| %EmbedDescription | The current embed point's description.                                 |
| %EmbedParameters  | The current embed point's current instance, as a comma-delimited list. |

Example:

```
#PREEMBED('! Before Embed Point: ' & %EmbedID & ' ' & %EmbedDescription & ' ' & |
 %EmbedParameters,%GenerateEmbedComments)
```

See Also: **#POSTEMBED**, **#EMPTYEMBED**

## Template Code Section Constraints

### #WHERE (define #CODE embed point availability)

#### #WHERE( *embeds* )

**#WHERE** Limits the availability of a #CODE to only those specific embedded source code points where the generated code would be appropriate.

*embeds* A comma-delimited list of #EMBED *identifiers* that specifies the embedded source code points that may use the #CODE to generate source code.

The **#WHERE** statement limits the availability of a #CODE to only those #EMBED embedded source code points where the generated code would be appropriate. A single #CODE may contain multiple #WHERE statements to explicitly define all the valid #EMBED embedded source code points. All the #WHERE statements in a #CODE are evaluated to determine which embedded source code points have been specifically enabled.

The *embeds* list must contain individual #EMBED *identifiers* delimited by commas. It may also contain ranges of embed points in the form *FirstIdentifier.LastIdentifier*, also delimited by commas. The *embeds* list may contain both types in a “mix and match” manner to define all suitable embedded source code points.

Example:

```
#CODE(ChangeProperty,'Change control property')
 #WHERE(%AfterWindowOpening..%CustomRoutines)
 #!Appropriate everywhere after window open
 #PROMPT('Control to change',CONTROL),%MyField,REQ
 #PROMPT('Property to change',@S20),%MyProperty,REQ
 #PROMPT('New Value',@S20),%MyValue,REQ
%MyField{%MyProperty} = '%MyValue'
```

See Also:           **#EMBED, #CODE, #RESTRICT**

## #RESTRICT (define section use constraints)

```
#RESTRICT [, WHERE(expression)]
 statements
#ENDRESTRICT
```

|                     |                                                                                                                                 |
|---------------------|---------------------------------------------------------------------------------------------------------------------------------|
| <b>#RESTRICT</b>    | Specifies conditions where a Template Code Section (#CODE, #CONTROL, #EXTENSION, #PROCEDURE, #PROGRAM, or #MODULE) can be used. |
| <b>WHERE</b>        | The #RESTRICT <i>statements</i> are executed only when the <i>expression</i> is true.                                           |
| <i>expression</i>   | A logical expression to limit execution of the #RESTRICT <i>statements</i> .                                                    |
| <i>statements</i>   | Template language code to #ACCEPT or #REJECT use of the section which contains the #RESTRICT structure.                         |
| <b>#ENDRESTRICT</b> | Terminates the #RESTRICT structure.                                                                                             |

The #RESTRICT structure provides a mechanism to limit the availability of a Template Code Section (#CODE, #CONTROL, #EXTENSION, #PROCEDURE, #PROGRAM, or #MODULE) at application design time to only those points where the generated code would be appropriate. Any WHERE clause on the Template Code Section is evaluated first, before #RESTRICT.

The #ACCEPT statement may be used to explicitly declare the section as appropriate for use. An implicit #ACCEPT also occurs if the #RESTRICT *statements* execute without encountering a #REJECT statement. The #REJECT statement must be used to specifically exclude the section from use. Both the #ACCEPT and #REJECT statements immediately terminate processing of the #RESTRICT code.

Example:

```
#CODE(ChangeControlSize,'Change control size')
#RESTRICT
 #CASE(%ControlType)
 #OF('LIST')
 #OROF('BUTTON')
 #REJECT
 #ELSE
 #ACCEPT
 #ENDCASE
#ENDRESTRICT
#PROMPT('Control to change',CONTROL),%MyField,REQ
#PROMPT('New Width',@n04),%NewWidth
#PROMPT('New Height',@n04),%NewHeight
%MyField{PROP:Width} = %NewWidth
%MyField{PROP:Height} = %NewHeight
```

See Also:

#ACCEPT, #REJECT

## #ACCEPT (section valid for use)

### #ACCEPT

The **#ACCEPT** statement terminates **#RESTRICT** processing, indicating that the Template Code Section (**#CODE**, **#CONTROL**, **#EXTENSION**, **#PROCEDURE**, **#PROGRAM**, or **#MODULE**) is valid.

The **#RESTRICT** structure contains Template language *statements* that evaluate the propriety of generating the section's source code. The **#ACCEPT** statement may be used to explicitly declare the section as appropriate. An implicit **#ACCEPT** also occurs if the **#RESTRICT** *statements* execute without encountering a **#REJECT** statement. The **#REJECT** statement must be used to specifically exclude the section from use. Both the **#ACCEPT** and **#REJECT** statements immediately terminate processing of the **#RESTRICT** code.

Example:

```
#CODE(ChangeControlSize,'Change control size')
#WHERE(%EventHandling)
#RESTRICT
#CASE(%ControlType)
#OF 'LIST'
#OROF 'BUTTON'
#REJECT
#ELSE
#ACCEPT
#ENDCASE
#ENDRESTRICT
#PROMPT('Control to change',CONTROL),%MyField,REQ
#PROMPT('New Width',@n04),%NewWidth
#PROMPT('New Height',@n04),%NewHeight
%MyField{PROP:Width} = %NewWidth
%MyField{PROP:Height} = %NewHeight
```

See Also: **#RESTRICT**, **#REJECT**

## #REJECT (section invalid for use)

---

### #REJECT

The **#REJECT** statement terminates **#RESTRICT** processing, indicating that the Template Code Section (**#CODE**, **#CONTROL**, **#EXTENSION**, **#PROCEDURE**, **#PROGRAM**, or **#MODULE**) is invalid.

The **#RESTRICT** structure contains Template language *statements* that evaluate the propriety of generating the section's source code. The **#ACCEPT** statement may be used to explicitly declare the section as appropriate. An implicit **#ACCEPT** also occurs if the **#RESTRICT** *statements* execute without encountering a **#REJECT** statement. The **#REJECT** statement must be used to specifically exclude the section from use. Both the **#ACCEPT** and **#REJECT** statements immediately terminate processing of the **#RESTRICT** code.

Example:

```
#CODE(ChangeControlSize,'Change control size')
#WHERE(%EventHandling)
#RESTRICT
#CASE(%ControlType)
#OF 'LIST'
#OROF 'BUTTON'
#REJECT
#ELSE
#ACCEPT
#ENDCASE
#ENDRESTRICT
#PROMPT('Control to change',CONTROL),%MyField,REQ
#PROMPT('New Width',@n04),%NewWidth
#PROMPT('New Height',@n04),%NewHeight
%MyField{PROP:Width} = %NewWidth
%MyField{PROP:Height} = %NewHeight
```

See Also: **#RESTRICT**, **#ACCEPT**



# 17 - DEFAULTS AND TEMPLATE DATA

## Default Data and Code

### #WINDOWS (default window structures)

```
#WINDOWS
 structures
#ENDWINDOWS
```

**#WINDOWS** Begins a default window data structure section.  
*structures* Default APPLICATION or WINDOW structures.  
**#ENDWINDOWS** Terminates the default window section.

The **#WINDOWS** structure contains default APPLICATION or WINDOW data structures for a procedure Template. The default window *structures* provide a starting point for the procedure's window design.

The **#WINDOWS** section may contain multiple *structures* which may be chosen as the starting point for the procedure's window design. If there is more than one window structure to choose from, the Application Generator displays a list of those *structures* available the first time the procedure's window is edited. The names of the windows which appear in the Application Generator's list comes from a preceding comment beginning with two exclamations and a right angle bracket (!!>).

If the procedure template contains a **#DEFAULT** procedure, there is no need for **#WINDOWS**, since the default window is already in the **#DEFAULT**. Therefore, the list does not appear when the window is first edited.

Example:

```
#WINDOWS
!!> Window
Label WINDOW('Caption'),AT(0,0,100,100)
 END
!!> Window with OK & Cancel
Label WINDOW('Caption'),AT(0,1,185,92)
 BUTTON('OK'),AT(144,10,35,14),DEFAULT,USE(?Ok)
 BUTTON('Cancel'),AT(144,28,36,14),USE(?Cancel)
 END
#ENDWINDOWS
```

## #REPORTS (default report structures)

```
#REPORTS
 structures
#ENDREPORTS
```

**#REPORTS** Begins a default report data structure section.

*structures* Default REPORT structures.

**#ENDREPORTS** Terminates the default report section.

The **#REPORTS** structure contains default REPORT data structures for a procedure Template. The default report *structures* provide a starting point for the procedure's report design.

The **#REPORTS** section may contain multiple *structures* which may be chosen as the starting point for the procedure's report design. If there is more than one report structure to choose from, the Application Generator displays a list of those *structures* available the first time the procedure's report is edited. The names of the windows which appear in the Application Generator's list comes from a preceding comment beginning with two exclamation points and a right angle bracket (!!>).

If the procedure template contains a **#DEFAULT** procedure, there is no need for **#REPORT**, since the default report is already in the **#DEFAULT**. Therefore, the list does not appear when the report is first edited.

Example:

```
#REPORTS
!!> Report
Label REPORT, AT(1000, 2500, 6000, 6000), THOUS
 HEADER, AT(1000, 1000, 6000, 1000)
 END
Detail DETAIL
 END
 FOOTER, AT(1000, 10000, 6000, 1000)
 END
 FORM, AT(1000, 1000, 6000, 9000)
 END
 END
#ENDREPORTS
```

## #LOCALDATA (default local data declarations)

---

```
#LOCALDATA
 declarations
#ENDLOCALDATA
```

**#LOCALDATA** Begins a default local data declaration section.  
*declarations* Data declarations.  
**#ENDLOCALDATA** Terminates the default local data declarations.

The **#LOCALDATA** structure contains default data *declarations* local to the procedure generated by the **#PROCEDURE** procedure Template. The *declarations* are then available to the programmer through the *Data* button on the **Procedure Properties** dialog. **#LOCALDATA** may only be placed in a **#PROCEDURE**, **#CODE**, **#CONTROL**, or **#EXTENSION** section of the Template. The *declarations* will appear in the generated procedure between the keywords **PROCEDURE** and **CODE**.

Example:

```
#LOCALDATA
Action BYTE !Disk action variable
TempFile CTRING(65) !Temporary filename variable
#ENDLOCALDATA
```

## #GLOBALDATA (default global data declarations)

---

```
#GLOBALDATA
 declarations
#ENDGLOBALDATA
```

**#GLOBALDATA** Begins a default global data declaration section.  
*declarations* Data declarations.  
**#ENDGLOBALDATA** Terminates the default global data declarations.

The **#GLOBALDATA** structure contains default data *declarations* global to the program. The *declarations* are then available to the programmer through the *Data* button on the **Global Properties** dialog. **#GLOBALDATA** may be placed in a **#PROGRAM**, **#PROCEDURE**, **#CODE**, **#CONTROL**, or **#EXTENSION** section of the Template. The *declarations* will appear in the global data section of the generated source code.

Example:

```
#GLOBALDATA
Action BYTE !Disk action variable
TempFile CTRING(65) !Temporary filename variable
#ENDGLOBALDATA
```

## #DEFAULT (default procedure starting point)

```
#DEFAULT
 procedure
#ENDEFAULT
```

**#DEFAULT** Begins a default procedure declaration section.  
*procedure* Default procedure in .TXA file format.  
**#ENDEFAULT** Terminates the default procedure declaration.

The **#DEFAULT** structure contains a single default *procedure* declaration in .TXA format as generated by the Application Generator's Export function. **#DEFAULT** may only be placed at the end of a **#PROCEDURE** section. You may have multiple **#DEFAULT** structures for a single **#PROCEDURE**. The enclosed *procedure* section of a .TXA file should contain a procedure of the preceding **#PROCEDURE**'s type. The recommended way to create a **#DEFAULT** structure is to edit the default procedure in the template registry, and then export the template as text, which creates a .TXA file.

Example:

```
#DEFAULT
NAME DefaultForm
[COMMON]
DESCRIPTION 'Default record update'
FROM Clarion Form
[PROMPTS]
%WindowOperationMode STRING ('Use WINDOW setting')
%INISaveWindow LONG (1)
[ADDITION]
NAME Clarion SaveButton
[INSTANCE]
INSTANCE 1
PROCPROP
[PROMPTS]
%InsertAllowed LONG (1)
%InsertMessage @S30 ('Record will be Added')
%ChangeAllowed LONG (1)
%ChangeMessage @S30 ('Record will be Changed')
%DeleteAllowed LONG (1)
%DeleteMessage @S30 ('Record will be Deleted')
%MessageHeader LONG (0)
[ADDITION]
NAME Clarion CancelButton
[INSTANCE]
INSTANCE 2
[WINDOW]
FormWindow WINDOW('Update Records...'),AT(18,5,289,159),CENTER,SYSTEM,GRAY,MDI
 BUTTON('OK'),AT(5,140,40,12),USE(?OK),DEFAULT,#SEQ(1),#ORIG(?OK),#LINK(?Cancel)
 BUTTON('Cancel'),AT(50,140,40,12),USE(?Cancel),#SEQ(2),#ORIG(?Cancel)
 STRING(@S40),AT(95,140,,),USE(ActionMessage)
END
#ENDEFAULT
```

## Symbol Management Statements

### #DECLARE (declare a user-defined symbol)

```
#DECLARE(symbol [, parentsymbol] [, type]) [, MULTI] [, UNIQUE] [, SAVE]
```

|                     |                                                                                                                                                                                                                                                                                                                       |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>#DECLARE</b>     | Explicitly declares a user-defined symbol.                                                                                                                                                                                                                                                                            |
| <i>symbol</i>       | The name of the symbol being declared. This must meet all the requirements of a user-defined symbol. This must not be a #PROMPT symbol or a variable in the same scope.                                                                                                                                               |
| <i>parentsymbol</i> | Specifies the parent of the <i>symbol</i> , indicating its value is dependent upon the current value in another symbol. This must be a multi-valued symbol. You may specify more than one <i>parentsymbol</i> if the <i>symbol</i> is dependent upon a set of symbols. This allows implicit multi-dimensional arrays. |
| <i>type</i>         | The data type of the <i>symbol</i> : LONG, REAL, or STRING. If omitted, the data type is STRING.                                                                                                                                                                                                                      |
| <b>MULTI</b>        | Specifies the <i>symbol</i> may contain multiple values.                                                                                                                                                                                                                                                              |
| <b>UNIQUE</b>       | Specifies a multi-valued <i>symbol</i> that cannot contain duplicate values. The values are stored in ascending order. This implicitly declares the symbol as multi-valued, the MULTI attribute is not required.                                                                                                      |
| <b>SAVE</b>         | Specifies the value(s) in the <i>symbol</i> are saved between source generation sessions. A <i>symbol</i> with the SAVE attribute may only be declared in the #APPLICATION area.                                                                                                                                      |

The **#DECLARE** statement explicitly declares a user-defined *symbol*. This may contain a single value or multiple values. All user-defined symbols must be explicitly declared with #DECLARE except those declared on a #PROMPT statement and #GROUP parameters.

The MULTI attribute declares the *symbol* as multi-valued. This allows the #FIX, #FOR, #ADD, #DELETE, #SELECT, and #FREE statements to operate on the *symbol*.

The UNIQUE attribute ensures all instances of a multi-valued *symbol* to be unique and sorted in ascending sequence. When UNIQUE is specified, MULTI is not required. The #ADD statement builds the *symbol* values in sorted order and only allows a single instance of every value in the *symbol* when each entry is added.

If the #DECLARE statement contains one or more *parentsymbol* parameters, the user-defined *symbol* is dependent on the *parentsymbols*. This means a

separate instance (or instances, if multi-valued) of the *symbol* is available for each instance of the *parentsymbol*. If there are no *parentsymbol* parameters, it is independent.

**#DECLARE** may be used to create dependent *symbols*. The *parentsymbol* must be a multi-valued symbol, whether it is a built-in or user-defined symbol.

The **SAVE** attribute causes a *symbol's* value(s) to be saved at the end of source generation and restored when the **#DECLARE** statement is executed at the beginning of the next source generation session. A *symbol* with the **SAVE** attribute may only be declared in the **#APPLICATION** section.

Example:

```
#APPLICATION('Sample One')
#DECLARE(%UserSymbol),SAVE #!Value saved after generation
 #! and restored for next generation
#DECLARE(%ModuleFile,%Module),UNIQUE,MULTI #!Level-1 dependent symbol
#DECLARE(%ModuleFilePut,%ModuleFile) #!Level-2 dependent symbol
#DECLARE(%ModuleFileDelete,%ModuleFile) #!Second Level-2 dependent symbol
```

See Also: **#FIX, #FOR, #ADD, #DELETE, #FREE**

## #ALIAS (access a symbol from another instance)

**#ALIAS**( *newsymbol* , *oldsymbol* [, *instance* ] )

|                  |                                                                                                                                                                                         |
|------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>#ALIAS</b>    | Declares a synonym for a user-defined symbol.                                                                                                                                           |
| <i>newsymbol</i> | Specifies the new synonym for the <i>oldsymbol</i> .                                                                                                                                    |
| <i>oldsymbol</i> | The name of the symbol for which to declare a synonym. This must meet all the requirements of a user-defined symbol. This must not be a #PROMPT symbol or a variable in the same scope. |
| <i>instance</i>  | An expression containing the instance of the addition containing the <i>oldsymbol</i> .                                                                                                 |

The **#ALIAS** statement declares a synonym for the user-defined *oldsymbol* declared in a #CODE, #CONTROL, or #EXTENSION template prompt for use in another.

Example:

```
#EXTENSION(GlobalSecurity,'Global Password Check'),APPLICATION
#DECLARE(%PasswordFile)
#DECLARE(%PasswordFileKey)

#EXTENSION(LocalSecurity,'Local Procedure Password Check'),PROCEDURE
#ALIAS(%PswdFile,%PasswordFile,%ControlInstance)
#ALIAS(%PswdFileKey,%PasswordFileKey,%ControlInstance)
```

See Also:               **#CODE, #CONTROL, #EXTENSION**

## #EQUATE (declare and assign value to a user-defined symbol)

---

**#EQUATE**( *symbol*, *value* )

|                |                                                                                   |
|----------------|-----------------------------------------------------------------------------------|
| <b>#EQUATE</b> | Declares and assigns a value to a single-valued user-defined symbol.              |
| <i>symbol</i>  | A single-valued user-defined symbol. This must not have been previously declared. |
| <i>value</i>   | A built-in or user-defined symbol, string constant, or an expression.             |

The **#EQUATE** statement declares the *symbol* and assigns the *value* to the *symbol*. This is directly equivalent to a **#DECLARE** statement followed by a **#SET** to assign it a value.

If the *value* parameter contains an expression, you may perform mathematics during source code generation. The expression may use any of the arithmetic, Boolean, and logical operators documented in the *Language Reference*. If the modulus division operator (%) is used in the expression, it must be followed by at least one blank space (to explicitly differentiate it from the Template symbols). Logical expressions always evaluate to 1 (True) or 0 (False). Clarion language procedure calls (those supported in **EVALUATE()**) and built-in template procedures are allowed.

Example:

```
#EQUATE(%NetworkApp,'Network')
#EQUATE(%MySymbol,%Primary)
```

See Also:           **#DECLARE**, **#SET**

## #ADD (add to multi-valued symbol)

**#ADD**( *symbol*, *expression* [, *position* ] )

|                   |                                                                                                                                                                                                                                                                                                             |
|-------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>#ADD</b>       | Adds a new instance to a multi-valued user-defined symbol.                                                                                                                                                                                                                                                  |
| <i>symbol</i>     | A multi-valued user-defined symbol.                                                                                                                                                                                                                                                                         |
| <i>expression</i> | An expression containing the value to place in the <i>symbol's</i> instance.                                                                                                                                                                                                                                |
| <i>position</i>   | An integer constant or symbol containing the instance number to add to the <i>symbol</i> . Instance numbering begins with one (1). If the <i>position</i> is greater than the number of previously existing instances plus one, the new instance is appended and no intervening instances are instantiated. |

The **#ADD** statement adds a value to a multi-valued user-defined *symbol*. An implied **#FIX** to that *symbol's* instance occurs. If the *symbol* is not a multi-valued user-defined symbol then a source generation error is produced.

If the *symbol* has been declared with the **UNIQUE** attribute, then the **#ADD** is a union operation into the existing set of *symbol's* values. Only one instance of the value being added may exist. Also, the **UNIQUE** attribute implies the **#ADD** is a sorted insert into the existing set of *symbol's* values. After each **#ADD**, all of the *symbol's* values will be in sorted order.

If the *symbol* has been declared without the **UNIQUE** attribute, duplicate values are allowed. The new value is added to the end of the list and may be a duplicate. If the *symbol* is a duplicate, then any dependent children instances are inherited.

Example:

```
#DECLARE(%ProcFilesPrefix),MULTI,UNIQUE #!Declare unique multi-valued symbol
#FIX(%File,%Primary) #!Build list of all file prefixes in proc
#ADD(%ProcFilesPrefix,%FilePre) #!Start with primary file
#FOR(%Secondary) #!Then add all secondary files
 #FIX(%File,%Secondary)
 #ADD(%ProcFilesPrefix,%FilePre)
#ENDFOR
```

See Also:

**#DECLARE**

## #DELETE (delete a multi-valued symbol instance)

**#DELETE**( *symbol* [, *position* ] )

|                 |                                                                                                                                                                                    |
|-----------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>#DELETE</b>  | Deletes the value from one instance of a multi-valued user-defined symbol.                                                                                                         |
| <i>symbol</i>   | A multi-valued user-defined symbol.                                                                                                                                                |
| <i>position</i> | An integer constant or symbol containing the instance number in the <i>symbol</i> . Instance numbering begins with one (1). If omitted, the default is the current fixed instance. |

The **#DELETE** statement deletes the value from one instance of a multi-valued user-defined symbol. If there are any symbols dependent upon the *symbol*, they are also cleared. If this is the last instance in the *symbol*, the instance is removed. You can get the current instance number to which a symbol is fixed by using the **INSTANCE(%symbol)** built-in template procedure.

Example:

```
#DECLARE(%ProcFilesPrefix),MULTI #!Declare multi-valued symbol
#ADD(%ProcFilesPrefix,'SAV') #!Add a value
#ADD(%ProcFilesPrefix,'BAK') #!Add a value
#ADD(%ProcFilesPrefix,'PRE') #!Add a value
#ADD(%ProcFilesPrefix,'QUE') #!Add a value

#DELETE(%ProcFilesPrefix,1) #!%ProcFilesPrefix contains: SAV, BAK, PRE, QUE
 #!Delete first value (SAV)
#FIX(%ProcFilesPrefix,'PRE') #!%ProcFilesPrefix contains: BAK, PRE, QUE
 #!Fix to a value
#DELETE(%ProcFilesPrefix) #!Delete it
 #!%ProcFilesPrefix contains: BAK, QUE
```

See Also:           **#DECLARE, #ADD**

## #DELETEALL (delete multiple multi-valued symbol instances)

**#DELETEALL**( *symbol*, *expression* )

**#DELETEALL** Deletes the values from specified instances of a multi-valued user-defined symbol.

*symbol* A multi-valued user-defined symbol.

*expression* An expression that defines the instances to delete.

The **#DELETEALL** statement deletes all values from the *symbol* that meet the *expression*.

Example:

```
#DECLARE(%ProcFilesPrefix),MULTI #!Declare multi-valued symbol
#ADD(%ProcFilesPrefix,'SAV') #!Add a value
#ADD(%ProcFilesPrefix,'BAK') #!Add a value
#ADD(%ProcFilesPrefix,'PRE') #!Add a value
#ADD(%ProcFilesPrefix,'BAK') #!Add a value
#ADD(%ProcFilesPrefix,'QUE') #!Add a value
 #!%ProcFilesPrefix contains: SAV, BAK, PRE, BAK, QUE
#DELETEALL(%ProcFilesPrefix,'BAK') #!Delete all BAK instances
 #!%ProcFilesPrefix now contains: SAV, PRE, QUE
```

See Also: **#DECLARE**, **#ADD**

## #PURGE (delete all single or multi-valued symbol instances)

### #PURGE( *symbol* )

**#PURGE** Deletes the values from all instances of a user-defined symbol.

*symbol* A user-defined symbol.

The **#PURGE** statement deletes all values from the *symbol*. If there are any symbols dependent upon the *symbol*, they are also cleared. If the *symbol* is dependent upon a multi-valued symbol, all instances of that dependent *symbol* are purged for all instances of the symbol upon which it is dependent.

Example:

```
#DECLARE(%ProcFilesPrefix),MULTI #!Declare multi-valued symbol
#ADD(%ProcFilesPrefix,'SAV') #!Add a value
#ADD(%ProcFilesPrefix,'BAK') #!Add a value
#ADD(%ProcFilesPrefix,'PRE') #!Add a value
#ADD(%ProcFilesPrefix,'BAK') #!Add a value
#ADD(%ProcFilesPrefix,'QUE') #!Add a value
 #!%ProcFilesPrefix contains: SAV, BAK, PRE, BAK, QUE
#PURGE(%ProcFilesPrefix) #!Delete all instances
```

See Also:           **#DECLARE, #ADD**

## #CLEAR (clear single-valued symbol)

### #CLEAR( *symbol* )

**#CLEAR** Removes the value from a single-valued user-defined symbol.

*symbol* A single-valued user-defined symbol.

The **#CLEAR** statement removes the value from a single-valued user-defined symbol. This statement is approximately the same as using **#SET** to assign a null value to the *symbol*, except it is more efficient.

Example:

```
#DECLARE(%SomeSymbol) #!Declare symbol
#SET(%SomeSymbol,'Value') #!Assign a value
 #!%SomeSymbol now contains: 'Value'
#CLEAR(%SomeSymbol) #!Clear value
 #!%SomeSymbol now contains: ''
```

See Also:           **#DECLARE, #ADD**

## #FREE (free a multi-valued symbol)

### #FREE( *symbol* )

**#FREE** Clears all instances of a multi-valued user-defined symbol.

*symbol* A multi-valued user-defined symbol.

The **#FREE** statement clears all instances of a multi-valued user-defined symbol. If there are any symbols dependent upon the *symbol*, they are also cleared.

Example:

```
#DECLARE(%ProcFilesPrefix),MULTI #!Declare multi-valued symbol
#ADD(%ProcFilesPrefix,'SAV') #!Add a value
#ADD(%ProcFilesPrefix,'BAK') #!Add a value
#ADD(%ProcFilesPrefix,'PRE') #!Add a value
#ADD(%ProcFilesPrefix,'BAK') #!Add a value
#ADD(%ProcFilesPrefix,'QUE') #!Add a value
 #!%ProcFilesPrefix contains: SAV, BAK, PRE, BAK, QUE
#DELETEALL(%ProcFilesPrefix,'BAK') #!Delete all BAK instances
 #!%ProcFilesPrefix now contains: SAV, PRE, QUE
#FREE(%ProcFilesPrefix) #!Free the symbol
 #!%ProcFilesPrefix now contains nothing
```

See Also: **#DECLARE, #ADD**

## #FIX (fix a multi-value symbol)

**#FIX**( *symbol*, *fixsymbol* )

|                  |                                                                          |
|------------------|--------------------------------------------------------------------------|
| <b>#FIX</b>      | Fixes a multi-valued symbol to the value of a single instance.           |
| <i>symbol</i>    | A multi-valued symbol.                                                   |
| <i>fixsymbol</i> | A symbol or expression containing the value to fix the <i>symbol</i> to. |

The **#FIX** statement fixes the current value of the multi-valued *symbol* to the value contained in the *fixsymbol*. This is done so that one instance of the *symbol* may be referenced outside a **#FOR** loop structure, or so you can reference the symbols dependent upon the multi-valued *symbol*.

The *fixsymbol* must contain a valid instance of one of the *symbol*'s multiple values. If the *fixsymbol* does not contain a valid instance, the *symbol* is cleared and contains no value when referenced. Unless **#ADD** has been used to add a new value and fix to that instance, **#FIX** or **#SELECT** must be used to set the value in a *symbol* before it contains any value for Template processing outside of a **#FOR** loop.

**#FIX** is completely independent of **#FOR** in that **#FOR** always loops through every instance of the *symbol*, whether there is a previous **#FIX** for that *symbol* or not. If there is a previous **#FIX** statement for that *symbol* before the **#FOR** loop, that *symbol* reverts to that previous *fixvalue* after the **#FOR** terminates.

If **#FIX** is used within a **#FOR** structure, the scope of the **#FIX** is limited to within the **#FOR** in which it is used. It does not change the **#FOR** symbol's iteration value if both the **#FOR** and **#FIX** happen to use the same symbol.

Example:

```
#SET(%OneFile,'HEADER') #! Put values into two User-defined symbols
#SET(%TwoFile,'DETAIL')
#FIX(%File,%OneFile) #! %File refers to 'HEADER'
#FOR(%File) #! %File iteratively refers to all file names
 #FIX(%File,%TwoFile) #! %File refers to 'DETAIL'
#ENDFOR #! %File refers to 'HEADER' again
```

See Also:           **#SELECT**

## #FIND (“super-fix” multi-value symbols)

**#FIND**( *symbol*, *fixsymbol* [, *limit* ] )

|                  |                                                                                           |
|------------------|-------------------------------------------------------------------------------------------|
| <b>#FIND</b>     | Fixes all multi-valued parent symbols to values that point to a single child instance.    |
| <i>symbol</i>    | A multi-valued symbol.                                                                    |
| <i>fixsymbol</i> | A symbol or expression containing the value to fix the <i>symbol</i> to.                  |
| <i>limit</i>     | A parent symbol which limits the search scope to the children of the <i>limit</i> symbol. |

The **#FIND** statement finds the first instance of the *fixsymbol* contained within the *symbol* then fixes it and all the “parent” symbols on which the *symbol* is dependent to the values that “point to” the value of the *fixsymbol* contained in the *symbol*. This is done so that all the symbol dependencies are aligned and you can reference other symbols dependent on “parent” symbols of the *symbol*.

For example, assume %ControlUse contains CUS:Name. The **#FIND(%Field,%ControlUse)** statement:

- Finds the first instance of %Field that matches the current value in %ControlUse (the first instance of CUS:Name in %Field) in the current procedure.
- Fixes %Field to that value (CUS:Name).
- Fixes %File to the name of the file containing that field (Customer).
- This allows the Template code to reference other the symbols dependent upon %File (like %FilePre to get the file’s prefix).

The *fixsymbol* must contain a valid instance of one of the *symbol*’s multiple values. If the *fixsymbol* does not contain a valid instance, the *symbol* is cleared and contains no value when referenced.

Example:

```
#FIND(%Field,%ControlUse) #!Fixes %Field and %File to %ControlUse parents
```

See Also:

**#SELECT**, **#FIX**

## #SELECT (fix a multi-value symbol)

---

**#SELECT**( *symbol*, *instance* )

|                 |                                                                      |
|-----------------|----------------------------------------------------------------------|
| <b>#SELECT</b>  | Fixes a multi-valued symbol to a particular <i>instance</i> number.  |
| <i>symbol</i>   | A multi-valued symbol.                                               |
| <i>instance</i> | An expression containing the number of the instance to which to fix. |

The **#SELECT** statement fixes the current value of the multi-valued *symbol* to a specific *instance*. The result of **#SELECT** is exactly the same as **#FIX**. Each *instance* in the multi-valued *symbol* is numbered starting with one (1).

The *instance* must contain a valid instance number of one of the *symbol*'s multiple values. If the *instance* is not valid, the *symbol* is cleared and contains no value when referenced. The **INSTANCE** built-in template procedure can return the instance number.

Unless **#ADD** has been used to add a new value and fix to that instance, **#FIX** or **#SELECT** must be used to set the value in a *symbol* before it contains any value for Template processing outside of a **#FOR** loop.

Example:

```
#SELECT(%File,1) #!Fix to first %File instance
```

## #POP (delete and re-fix a multi-value symbol)

### #POP( *symbol* )

**#POP** Deletes the last instance of a multi-valued symbol and re-fixes to the new last instance.

*symbol* A multi-valued user-defined symbol.

The **#POP** statement deletes the last instance of a multi-valued symbol and re-fixes to the new last instance. This is directly equivalent to issuing a **#DELETE(%MultiSymbol,ITEMS(%MultiSymbol))** statement followed by **#SELECT(%MultiSymbol,ITEMS(%MultiSymbol))** statement.

Example:

```
#POP(%MultiSymbol) #!Delete last instance and fix to new last instance
```

## #SET (assign value to a user-defined symbol)

### #SET( *symbol,value* )

**#SET** Assigns a value to a single-valued user-defined symbol.

*symbol* A single-valued user-defined symbol. This must have been previously declared with the **#DECLARE** statement.

*value* A built-in or user-defined symbol, string constant, or an expression.

The **#SET** statement assigns the *value* to the *symbol*. If the *value* parameter contains an expression, you may perform mathematics during source code generation. The expression may use any of the arithmetic, Boolean, and logical operators documented in the *Language Reference*. If the modulus division operator (%) is used in the expression, it must be followed by at least one blank space (to explicitly differentiate it from the Template symbols). Logical expressions always evaluate to 1 (True) or 0 (False). Clarion language procedure calls (those supported in **EVALUATE()**) and built-in template procedures are allowed.

Example:

```
#SET(%NetworkApp,'Network')
#SET(%MySymbol,%Primary)
#FOR(%File)
 #SET(%FilesCounter,%FilesCounter + 1)
%FileStructure
#ENDFOR
```

See Also: **#DECLARE**



# 18 - PROGRAMMER INPUT

## Input and Validation Statements

### #PROMPT (prompt for programmer input)

```
#PROMPT(string, type) [, symbol] [, REQ] [, DEFAULT(default)] [, ICON(file)] [, AT()]
[, PROMPTAT()] [, MULTI(description)] [, UNIQUE] [, INLINE] [, VALUE(value)]
[, SELECTION(description)] [, CHOICE] [, WHENACCEPTED(expression)]
[, HSCROLL] [, SORT]
```

|                 |                                                                                                                                                                                                                                                                                          |
|-----------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>#PROMPT</b>  | Asks the programmer for input.                                                                                                                                                                                                                                                           |
| <i>string</i>   | A string constant containing the text to display as the input prompt. This may contain an ampersand (&) denoting a “hot” key used in conjunction with the ALT key to get to this field on the properties screen.                                                                         |
| <i>type</i>     | A picture token or prompt keyword.                                                                                                                                                                                                                                                       |
| <i>symbol</i>   | A User-defined symbol to receive the input. A #PROMPT with a RADIO or EMBED <i>type</i> cannot have a <i>symbol</i> , all other <i>types</i> must have a <i>symbol</i> .                                                                                                                 |
| <b>REQ</b>      | Specifies the prompt cannot be left blank or zero.                                                                                                                                                                                                                                       |
| <b>DEFAULT</b>  | Specifies an initial value (which may be overridden).                                                                                                                                                                                                                                    |
| <i>default</i>  | A string constant containing the initial value.                                                                                                                                                                                                                                          |
| <b>ICON</b>     | Specifies an icon for the button face of a #PROMPT with the MULTI attribute.                                                                                                                                                                                                             |
| <i>file</i>     | A string constant containing the name of the .ICO file to display on the button face.                                                                                                                                                                                                    |
| <b>AT</b>       | Specifies the position of the prompt entry area in the window, relative to the first prompt placed on the window from the Template (excluding the standard prompts on every procedure properties window). This attribute takes the same parameters as the Clarion language AT attribute. |
| <b>PROMPTAT</b> | Specifies the position of the prompt <i>string</i> relative to the first prompt placed on the window from the Template (excluding the standard prompts on every procedure properties window). This attribute takes the same parameters as the Clarion language AT attribute.             |
| <b>MULTI</b>    | Specifies you may enter multiple values for the #PROMPT. The prompt appears as a button which pops up a list box allowing the programmer to enter multiple values, unless the INLINE attribute is also present.                                                                          |

|                     |                                                                                                                                                                                                                                                                                          |
|---------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>description</i>  | A string constant containing the name to display on the button face and at the top of the list of prompt values.                                                                                                                                                                         |
| <b>UNIQUE</b>       | The multiple values the programmer enters for the #PROMPT are unique values and are sorted in ascending order. The MULTI attribute must also be present.                                                                                                                                 |
| <b>INLINE</b>       | The multiple values the programmer enters for the #PROMPT appears as a list box with update buttons which allow the programmer to enter multiple values. The MULTI attribute must also be present.                                                                                       |
| <b>VALUE</b>        | Specifies the value assigned to the <i>symbol</i> when the #PROMPT is selected. Valid only when the #PROMPT <i>type</i> is RADIO.                                                                                                                                                        |
| <i>value</i>        | A string constant containing the value to assign to the <i>symbol</i> .                                                                                                                                                                                                                  |
| <b>SELECTION</b>    | Specifies the programmer may select multiple values for the #PROMPT from the list of choices presented by the FROM <i>type</i> . The prompt appears as a button which pops up a list box allowing the programmer to choose multiple values, unless the INLINE attribute is also present. |
| <b>CHOICE</b>       | Specifies the <i>symbol</i> receives the ordinal position number of the selection instead of the <i>string</i> text from a DROP or OPTION/RADIO #PROMPT set.                                                                                                                             |
| <b>WHENACCEPTED</b> | Specifies an <i>expression</i> to execute when the #PROMPT posts an Accepted event (the programmer has entered or chosen data).                                                                                                                                                          |
| <i>group</i>        | The name of the #GROUP to execute.                                                                                                                                                                                                                                                       |
| <b>HSCROLL</b>      | Specifies horizontal scroll bars in the DROP list. Valid only when the #PROMPT <i>type</i> is DROP.                                                                                                                                                                                      |
| <b>SORT</b>         | Specifies alphabetically sorted display of items in the droplist. Valid only when the #PROMPT <i>type</i> is FROM.                                                                                                                                                                       |

The **#PROMPT** statement asks the programmer for input. A #PROMPT statement may be placed in #APPLICATION, #PROCEDURE, #CODE, #CONTROL, #EXTENSION, #UTILITY, or #FIELD sections. It may not be placed in a #PROGRAM, #MODULE, #TEMPLATE, or #GROUP section. When the #PROMPT is placed in a template section, the prompt *string* and its associated entry field are placed as follows:

| <u>Section Name</u> | <u>Window Name</u>             |
|---------------------|--------------------------------|
| #APPLICATION        | Global Settings                |
| #PROCEDURE          | Procedure Properties           |
| #CODE               | Embeds Dialog                  |
| #CONTROL            | Control Properties Actions Tab |
| #EXTENSION          | Extensions Dialog              |
| #FIELD              | Control Properties Actions Tab |

The *type* parameter may either contain a picture token to format the programmer's input, or one of the following keywords:

|            |                                                                |
|------------|----------------------------------------------------------------|
| PROCEDURE  | The label of a procedure                                       |
| FILE       | The label of a data file                                       |
| KEY        | The label of a key (can be limited to one file)                |
| COMPONENT  | The label of a key component field (can be limited to one key) |
| FIELD      | The label of a file field (can be limited to one file)         |
| EXPR       | A multi-field selection box that builds an expression          |
| OPTFIELD   | Constant text or the label of a file field                     |
| FORMAT     | Calls the listbox formatter.                                   |
| PICTURE    | Calls the picture token formatter.                             |
| DROP       | Creates a droplist of items specified in its parameter         |
| KEYCODE    | A keycode or keycode EQUATE                                    |
| OPTION     | Creates a radio button structure                               |
| RADIO      | Creates a radio button                                         |
| CHECK      | Creates a check box                                            |
| CONTROL    | A window control                                               |
| FROM       | Creates a droplist of items contained in its symbol parameter  |
| EMBED      | Allows the user to edit a specified embedded source code point |
| SPIN       | Creates a spin control                                         |
| TEXT       | Creates a text entry control                                   |
| OPENDIALOG | Calls a standard Windows Open File dialog                      |
| SAVEDIALOG | Calls a standard Windows Save File dialog                      |
| COLOR      | Calls a standard Windows Color dialog                          |

For all *types* except RADIO and CHECK (and MULTI attribute prompts), the #PROMPT *string* is displayed on the screen immediately to the left of its data input area. A #PROMPT with the REQ attribute cannot be left blank or zero; it is a required input field. The DEFAULT attribute may be used to provide the programmer with an initial *value* in the #PROMPT, which may be overridden at design time.

A #PROMPT with a RADIO *type* creates one Radio button for the immediately preceding #PROMPT with an OPTION *type*. There may be multiple RADIOS for one OPTION. Each RADIO's *string*, when selected, is placed in the closest preceding OPTION's *symbol*. The OPTION structure is terminated by the first #PROMPT following it that is not a RADIO.

The MULTI attribute specifies the programmer may enter multiple values for the #PROMPT. A button appears on the Properties window with the *description* on its face. Alternatively, this can have an ICON attribute to name an .ICO file to display on the button face. This button calls a window containing a list box to display all the multiple values entered for the #PROMPT, along with Insert, Change, and Delete buttons. These three buttons call another window containing the #PROMPT *string* and its data entry field to allow the programmer to update the entries in the list.

When the programmer has entered a value for the #PROMPT, the input value is assigned to the *symbol*. The value entered by the programmer may be checked for validity by one or more #VALIDATE statements immediately following the #PROMPT statement.

The value(s) placed in the *symbol* may be used or evaluated elsewhere within the Template. A *symbol* defined by a #PROMPT in the #APPLICATION section of the Template is Global, it can be used or evaluated anywhere in the Template. A *symbol* defined by #PROMPT in a #PROCEDURE section is Local, and is a dependent symbol to %Procedure; it can be used or evaluated only within that #PROCEDURE section. A *symbol* defined by #PROMPT in a #CODE, #CONTROL, or #EXTENSION section of the Template can be used or evaluated only within that section.

Example:

```
#PROMPT('Ask for Input',@s20),%InputSymbol #!Simple input
#PROMPT('Ask for FileName',FILE),%InputFile,REQ #!Required filename
#PROMPT('Pick One',OPTION),%InputChoice #!Mutually exclusive options
#PROMPT('Choice One',RADIO)
#PROMPT('Choice Two',RADIO)
#PROMPT('Next Procedure',PROCEDURE),%NextProc #!Prompt for procedure name
#PROMPT('Ask for Multiple Input',@s20),%MultiSymbol,MULTI('Input Values...')
 #!Prompt for multiple input
```

See Also:

#DISPLAY, #VALIDATE, #GROUP, #BOXED, #ENABLE, #BUTTON

## #VALIDATE (validate prompt input)

---

**#VALIDATE( *expression*,*message* )**

|                   |                                                                                   |
|-------------------|-----------------------------------------------------------------------------------|
| <b>#VALIDATE</b>  | Validates the data entered into the immediately preceding #PROMPT field.          |
| <i>expression</i> | The expression to use to validate the entered data.                               |
| <i>message</i>    | A string constant containing the error message to display if the data is invalid. |

The **#VALIDATE** statement validates the data entered into the #PROMPT field immediately preceding the #VALIDATE. The *expression* is evaluated when the OK button is pressed on the Procedure Properties window. If the *expression* is false, the *message* is displayed to the programmer in a message box, and control is given to the #PROMPT field that immediately precedes the #VALIDATE. There may be multiple #VALIDATE statements following a #PROMPT to validate the entry.

Example:

```
#PROMPT('Input Value, Even numbers from 100-200',@N3),%Value
#VALIDATE((%Value > 100) AND (%Value < 200),'Value must be between 100 and 200')
#VALIDATE((%Value % 2 = 0),'Value must be an even number')
#PROMPT('Screen Field',CONTROL),%SomeField
#VALIDATE(%ScreenFieldType = 'LIST','Must select a list box')
```

See Also:

**#PROMPT**

## #ENABLE (enable/disable prompts)

```
#ENABLE(expression) [, CLEAR] [, SECTION]
 prompts
#ENDENABLE
```

|                   |                                                                                                                        |
|-------------------|------------------------------------------------------------------------------------------------------------------------|
| <b>#ENABLE</b>    | Begins a group of <i>prompts</i> which may be enabled or disabled based upon the evaluation of the <i>expression</i> . |
| <i>expression</i> | The expression which controls the prompt enable/disable.                                                               |
| <b>CLEAR</b>      | Specifies the <i>prompts</i> symbol values are cleared when disabled.                                                  |
| <b>SECTION</b>    | Specifies all AT() attributes for the <i>prompts</i> are positioned relative to the start of the #ENABLE section.      |
| <i>prompts</i>    | One or more #PROMPT, #BUTTON, #DISPLAY, #ENABLE, and/or #VALIDATE statements.                                          |
| <b>#ENDENABLE</b> | Terminates the group of <i>prompts</i> .                                                                               |

The **#ENABLE** structure contains *prompts* which may be enabled or disabled based upon the evaluation of the *expression*. If the *expression* is true, the *prompts* are enabled, otherwise they are disabled. The *prompts* appear dimmed when disabled and the programmer may not enter data in them.

Example:

```
#PROMPT('Pick One',OPTION),%InputChoice #!Mutually exclusive options
#PROMPT('Choice One',RADIO)
#PROMPT('Choice Two',RADIO)
#ENABLE(%InputChoice = 'Choice Two')
 #PROMPT('Screen Field',CONTROL),%SomeField #!Enabled only for Choice Two
 #VALIDATE(%ScreenFieldType = 'LIST','Must select a list box')
#ENDENABLE
```

See Also:           **#PROMPT, #GROUP, #BOXED, #BUTTON**

## #BUTTON (call another page of prompts)

```
#BUTTON(string [, icon]) [, HLP(id)] [, AT()] [, REQ] [, INLINE] [, WHENACCEPTED(group)]
 [, | FROM(multisymbol, expression)] [, WHERE(condition)] |] [, SORT]
 | MULTI(fromsymbol, expression) |
 prompts
#ENDBUTTON
```

|                   |                                                                                                                                                                                                                                                                                                                                                                              |
|-------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>#BUTTON</b>    | Creates a command button to call another page of <i>prompts</i> .                                                                                                                                                                                                                                                                                                            |
| <i>string</i>     | An expression containing the text to display on the button's face. This may contain an ampersand (&) to indicate the "hot" letter for the button.                                                                                                                                                                                                                            |
| <i>icon</i>       | A string constant containing the name of an .ICO file or standard icon to display on the button's face. The <i>string</i> then serves only for "hot" key definition.                                                                                                                                                                                                         |
| <b>HLP</b>        | Specifies on-line help is available for the #BUTTON.                                                                                                                                                                                                                                                                                                                         |
| <i>id</i>         | A string constant containing the identifier to access the Help system. This may be either a Help keyword or "context string."                                                                                                                                                                                                                                                |
| <b>AT</b>         | Specifies the position of the button in the window, relative to the first prompt placed on the window from the Template (excluding the standard prompts on every procedure properties window). This attribute takes the same parameters as the Clarion language AT attribute. Specifying zero width and height indicates the #BUTTON occupies no space on the prompt dialog. |
| <b>REQ</b>        | Specifies the programmer must press the button at least once when the procedure is created.                                                                                                                                                                                                                                                                                  |
| <b>FROM</b>       | Specifies the programmer may enter a set of values for the <i>prompts</i> for each instance of the <i>fromsymbol</i> .                                                                                                                                                                                                                                                       |
| <i>fromsymbol</i> | A built-in multi-valued symbol which pre-defines all instances on which the <i>prompts</i> symbols are dependent. The programmer may not add, change, or delete instances of the <i>fromsymbol</i> .                                                                                                                                                                         |
| <i>expression</i> | A string expression to format data display in the multiple value display list box.                                                                                                                                                                                                                                                                                           |
| <b>WHERE</b>      | Specifies the #BUTTON displays only those instances of the <i>fromsymbol</i> where the <i>condition</i> is true.                                                                                                                                                                                                                                                             |
| <i>condition</i>  | An expression that specifies the condition for use.                                                                                                                                                                                                                                                                                                                          |
| <b>MULTI</b>      | Specifies the programmer may enter multiple sets of values for the prompts. This allows multiple related groups of prompts.                                                                                                                                                                                                                                                  |

|                     |                                                                                                                                                                                                            |
|---------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>multisymbol</i>  | A user-defined symbol on which all the <i>prompts</i> symbols are dependent. This symbol is internally assigned a unique value for each set of <i>prompts</i> .                                            |
| <b>INLINE</b>       | The multiple values the programmer enters for the #BUTTON appears as a list box with update buttons which allow the programmer to enter multiple values. The MULTI or FROM attribute must also be present. |
| <b>WHENACCEPTED</b> | Specifies the named <i>group</i> to execute when the #BUTTON is pressed.                                                                                                                                   |
| <i>group</i>        | The name of the #GROUP to execute.                                                                                                                                                                         |
| <b>SORT</b>         | Specifies alphabetically sorted display of items in the droplist. Valid only when the FROM attribute is present.                                                                                           |
| <i>prompts</i>      | One or more #PROMPT statements. This may also contain #DISPLAY, #VALIDATE, #ENABLE, and #BUTTON statements.                                                                                                |
| <b>#ENDBUTTON</b>   | Terminates the group of <i>prompts</i> which are on the page called by the #BUTTON.                                                                                                                        |

The #BUTTON statement creates a command button displaying either the *string* or the *icon* on its face. When the programmer presses the button, a new page of *prompts* appears for selection and entry.

Each new page of *prompts* has its own OK, CANCEL, and TEMPLATE HELP buttons as standard fields. All other fields on the page are generated from the *prompts* within the #BUTTON structure.

Each page's OK button closes the current page of *prompts*, saving the data the programmer entered in the *prompts*, then returns to the prior window. The CANCEL button closes the current page of *prompts* without saving, then returns to the prior window. If the page calls another page with a nested #BUTTON statement and the programmer presses OK on the lowest level page, then CANCEL on the page that called it, the entire transaction is cancelled.

The MULTI attribute specifies the programmer may enter multiple sets of values for the *prompts*. The button calls a window containing a list box to display all the multiple values entered for the sets of *prompts*, along with Insert, Change, and Delete buttons. These three buttons call another window containing all the *prompts* to allow the programmer to update the entries in the list. The *expression* is used to format the information for display in the list box.

The FROM attribute also specifies the programmer may enter multiple sets of values for the *prompts*. The button calls a window containing a list box that displays each instance of the *fromsymbol*, along with an Edit button. This button calls another window containing all the *prompts* to allow the

programmer to update the entries associated with that instance of the *fromsymbol*. The *expression* is used to format the information for display in the list box. The WHERE attribute may be used to limit the instances of the *fromsymbol* to only those that meet the WHERE *condition*.

Example:

```
#PROMPT('Name a File',FILE),%FileName #!Prompt on the first page

#BUTTON('Page Two') #!Button on first page calls
 #PROMPT('Pick One',OPTION),%InputChoice #!These prompts on second page
 #PROMPT('Choice One',RADIO)
 #PROMPT('Choice Two',RADIO)
 #ENABLE(%InputChoice = 'Choice Two')
 #PROMPT('Screen Field',CONTROL),%SomeField
 #VALIDATE(%ScreenFieldType = 'LIST','Must select a list box')
 #ENDENABLE
#ENDBUTTON #!Terminate second page prompts

#PROMPT('Enter some value',@S20),%InputValue1 #!Another prompt on first page

 #!Multiple input button:
#BUTTON('Multiple Names'),MULTI(%ButtonSymbol,%ForeName & ' ' & %SurName)
 #PROMPT('First Name',@S20),%ForeName
 #PROMPT('Last Name',@S20),%SurName
#ENDBUTTON #!Terminate second page prompts

#PROMPT('Enter another value',@S20),%InputValue2 #!Another prompt on first page
 #!Multiple input button dependent on %File:
#BUTTON('File Options'),FROM(%File)
 #PROMPT('Open Access Mode',DROP('Open|Share'),%FileOpenMode
#ENDBUTTON #!Terminate second page prompts
```

See Also: **#PROMPT, #VALIDATE, #ENABLE**

## #WITH (associate prompts with a symbol instance)

```
#WITH(symbol , value)
 prompts
#ENDWITH
```

|                 |                                                                                                                                                                              |
|-----------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>#WITH</b>    | Allows a set of prompts to be associated with a single instance of a <i>symbol</i> (as specified by the <i>value</i> expression).                                            |
| <i>symbol</i>   | A previously declared unique multi-valued template symbol.                                                                                                                   |
| <i>value</i>    | A string expression containing a unique value to add to the <i>symbol</i> .                                                                                                  |
| <i>prompts</i>  | One or more #PROMPT statements whose values become dependent on the value of the <i>symbol</i> . This may also contain #DISPLAY, #VALIDATE, #ENABLE, and #BUTTON statements. |
| <b>#ENDWITH</b> | Terminates the group of <i>prompts</i> associated with the #WITH <i>symbol</i> .                                                                                             |

The **#WITH** statement adds a new instance to the *symbol* (as specified by the *value* expression) and makes the values of all the *prompts* dependent upon that instance. #WITH allows the same set of prompts, with the same prompt symbols, to be used more than once on the same screen.

Example:

```
#WITH(%ClassItem,'Default')
 #INSERT(%ClassPrompts)
#ENDWITH
```

See Also:

#PROMPT, #VALIDATE, #ENABLE, #DISPLAY, #BUTTON

## #FIELD (control prompts)

```
#FIELD, WHERE(expression)
 prompts
#ENDFIELD
```

|                   |                                                                                            |
|-------------------|--------------------------------------------------------------------------------------------|
| <b>#FIELD</b>     | Begins a control prompts section.                                                          |
| <b>WHERE</b>      | Specifies the #FIELD is used only for those instances where the <i>expression</i> is true. |
| <i>expression</i> | An expression that specifies the condition for use.                                        |
| <i>prompts</i>    | Prompt (#PROMPT, #BUTTON, etc.) statements.                                                |
| <b>#ENDFIELD</b>  | Terminates the section.                                                                    |

The **#FIELD** structure contains *prompts* for controls that are populated onto a window. These *prompts* appear in the Actions... dialog.

The list of field prompts appearing in the Actions... dialog is built in the following manner:

1. #CONTROL prompts.
2. #PROCEDURE-level #FIELD prompts (also from inserted #GROUPs).
3. #PROCEDURE-level #FIELD prompts from active #EXTENSION sections.
4. #CONTROL-level #FIELD prompts.
5. #CODE-level #FIELD prompts.

The values the user inputs into the #FIELD prompts are used to generate the source to govern the behavior of the control.

Example:

```
#FIELD, WHERE(%ControlType = 'BUTTON')
 #PROMPT('Enter procedure call',PROCEDURE),%ButtonProc
#ENDFIELD
```

See Also:

#PROMPT, #VALIDATE, #ENABLE, #DISPLAY, #BUTTON

## #PREPARE (setup prompt symbols)

```
#PREPARE
 statements
#ENDPREPARE
```

**#PREPARE** Begins a prompts symbol setup section.

*statements* Template language statements to fix multi-valued symbols to the values needed to process the #PROMPT or #BUTTON statement preceding the #PREPARE.

**#ENDPREPARE** Terminates the section.

The **#PREPARE** structure contains Template language statements to fix multi-valued symbols to the values needed to process the #PROMPT or #BUTTON statements preceding the #PREPARE.

A #PREPARE section that precedes all the prompts is executed once, before the prompts are added to a window. In particular, if the prompts are displayed on the procedure properties screen, editing the window definition and returning does not execute the prepare code.

Example:

```
#CODE (SetProperty, 'Set a property on a control')
#PREPARE
 #DECLARE(%Choices),MULTI
 #DECLARE(%NextLine)
 #FREE(%Choices)
 #OPEN('property.clw'),READ
 #READ(%NextLine)
 #LOOP WHILE(%NextLine <> %Eof)
 #! Exclude PROPLIST: properties
 #IF (SUB(%NextLine, 1, 5) = 'PROP:')
 #ADD(%Choices, SUB(%NextLine, 1, INSTRING(' ',%NextLine)-1))
 #END
 #READ (%NextLine)
 #END
 #CLOSE,READ
#ENDPREPARE
#PROMPT('Control:', FROM(%Control)),%Target
#PROMPT('Property:', FROM(%Choices)),%Selection
#PROMPT('Value:', @s255),%TargetValue
%target{%Selection} = %TargetValue
```

## #PROMPT Entry Types

### CHECK (check box)

#### CHECK

The **CHECK** type in a #PROMPT statement indicates the prompt's *symbol* is a toggle switch which is used only for on/off, yes/no, or true/false evaluation. CHECK puts a check box on screen in the entry area for the #PROMPT. When the Check box is toggled on, the prompt's *symbol* equals %True. When the Check box is toggled off, the prompt's *symbol* equals %False.

Example:

```
#PROMPT('Network Application',CHECK),%NetworkApp
```

### COLOR (call Color dialog)

#### COLOR

The **COLOR** type in a #PROMPT statement indicates the prompt's *symbol* must contain the name of the color selected by the programmer from the Windows standard Color dialog.

Example:

```
#PROMPT('Ask for Color',COLOR),%ColorSymbol
```

### COMPONENT (list of KEY fields)

#### COMPONENT [ ( *scope* ) ]

**COMPONENT** Displays a list of KEY component fields.

*scope* A symbol containing a KEY. If omitted, the list displays all KEY components for all KEYS in all FILES.

The **COMPONENT** type in a #PROMPT statement indicates the prompt's *symbol* must contain the label of one of the component fields of a KEY. A list of available KEY fields pops up when the #PROMPT is encountered on the Properties screen.

The COMPONENT may have a *scope* parameter that limits the KEY components available in the list. If *scope* is the label of a KEY, the list displays all KEY components for that KEY.

Example:

```
#PROMPT('Record Selector',COMPONENT(%Primary)),%RecordSelector
```

## CONTROL (list of window fields)

### CONTROL

The **CONTROL** *type* in a #PROMPT statement indicates the prompt's *symbol* must contain the field equate label of a window control. A list of available controls pops up when the #PROMPT is encountered on the Properties screen.

Example:

```
#PROMPT('Locator Field',CONTROL),%Locator
```

## DROP (droplist of items)

### DROP [ ( *scope* ) ]

|              |                                                                                                                                   |
|--------------|-----------------------------------------------------------------------------------------------------------------------------------|
| <b>DROP</b>  | Creates a droplist of items.                                                                                                      |
| <i>scope</i> | A string constant or expression containing the items for the list. Each item must be delimited by the vertical bar ( ) character. |

The **DROP** *type* in a #PROMPT statement indicates the prompt's *symbol* must contain one item from the list specified in the *scope* parameter.

The *scope* parameter must contain all the items for the list. The *scope* may optionally contain text in square brackets following each choice that specifies the value to assign to the #PROMPT's *symbol*. This is useful for internationalization.

The list of items drops down just like a Clarion language LIST control with the DROP attribute. If no default value is specified, the prompt's symbol defaults to the first value in the *scope* list.

Example:

```
#PROMPT('If file not found',DROP('Create the file|Halt Program')),%FileNotFound
```

```
#PROMPT('Que es esto?',DROP('Nombre[Name]|Numero[Number]')),%WhatIsThis
#!"Name" is the value put in %WhatIsThis when the user chooses "Nombre"
#!"Number" is the value put in %WhatIsThis when the user chooses "Numero"
```

## EMBED (enter embedded source)

**EMBED**( *identifier* [, *instance* ] )

|                   |                                                                                                                                                                                                                                      |
|-------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>EMBED</b>      | Specifies the prompt directly edits an embedded source code point.                                                                                                                                                                   |
| <i>identifier</i> | The user-defined template symbol which identifies the #EMBED embedded source code point to edit.                                                                                                                                     |
| <i>instance</i>   | A string constant or expression containing one of the values in the multi-valued symbol used by the #EMBED. You must have as many <i>instances</i> as are necessary to explicitly identify the single #EMBED point instance to edit. |

The **EMBED** *type* in a #PROMPT statement indicates the prompt is used to directly edit an embedded source code point. This places an an entry area with an ellipsis (...) button next to the prompt to allow the user access to the embedded source code point. The programmer may enter a procedure call in the entry area, or press the ellipsis (...) button to go into the normal source dialog.

If the #EMBED is associated with a multi-valued symbol, you must identify the specific *instance* of the #EMBED. If you use a multi-valued symbol as an *instance* expression, it must be fixed to a single value. Most commonly, this would be used in a #FIELD structure.

Example:

```
#PROMPT('Embedded Data Declarations',EMBED(%DataSection))
#FIELD, WHERE(%ControlType = 'BUTTON')
 #PROMPT('Action when button is pressed',EMBED(%ControlEvent,%Control,'Accepted'))
#ENDFIELD
```

## EXPR (appended data fields)

**EXPR**

The **EXPR** *type* in a #PROMPT statement indicates the prompt's *symbol* will contain an expression concatenating multiple fields in a data file. When a field is selected, the field name is appended to the existing *symbol* contents (Field1 & Field2 & Field3). A list of available fields pops up when the programmer presses the "...” button next to the #PROMPT entry box on the Properties screen.

Example:

```
#PROMPT('Default Value',EXPR),%DefaultValue
```

## FIELD (list of data fields)

---

**FIELD** [ ( *scope* ) ]

**FIELD**

Displays a list of fields in FILES.

*scope*

A symbol containing a FILE label. If omitted, the list displays all fields for all FILES.

The **FIELD** *type* in a #PROMPT statement indicates the prompt's *symbol* must contain the label of a field in a data file. A list of available fields pops up when the #PROMPT is encountered on the Properties screen.

There may be a *scope* parameter that limits the fields available in the list. If *scope* names a FILE, the list displays all fields in the FILE. If there is no *scope* parameter, the list displays all fields in all FILES.

Example:

```
#PROMPT('Locator Field',FIELD(%Primary)),%Locator
```

## FILE (list of files)

---

**FILE**

The **FILE** *type* in a #PROMPT statement indicates the prompt's *symbol* must contain the label of a data file. A list of available files from the procedure's File Schematic pops up when the #PROMPT is encountered on the Properties screen.

Example:

```
#PROMPT('Logout File',FILE),%LogoutFile
```

## FORMAT (call listbox formatter)

### FORMAT

The **FORMAT** *type* in a #PROMPT statement indicates the prompt's *symbol* must contain a LIST or COMBO control's FORMAT attribute string, so it calls the listbox formatter to create it.

Example:

```
#PROMPT('Alternate LIST format',FORMAT),%AlternateFormatString
```

## FROM (list of symbol values)

### FROM( *symbol* [, *expression* ] [, *value* ] )

|                   |                                                                                                                                                                      |
|-------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>FROM</b>       | Specifies a drop-down list of values from the <i>symbol</i> .                                                                                                        |
| <i>symbol</i>     | A multi-valued symbol (built-in or user-defined).                                                                                                                    |
| <i>expression</i> | An expression which controls which <i>symbol</i> values are displayed. Only <i>symbol</i> values where the <i>expression</i> is true are displayed in the drop list. |
| <i>value</i>      | The symbol containing the values to display for the prompt and assigned into the <i>symbol</i> .                                                                     |

The **FROM** *type* in a #PROMPT statement indicates the user must select one item from the list contained in the *symbol*. The *expression* can be used to limit the *values* displayed, while the *value* defines the display elements.

Example:

```
#PROMPT('Select an Event',FROM(%ControlEvent)),%WhichEvent
#PROMPT('Select a Button',FROM(%ControlField,%ControlType = 'BUTTON')),%WhichButton
#PROMPT('Pick a Field',FROM(%Control,%ControlUse <> '',%ControlUse)),%MyButton
```

## KEY (list of keys)

---

### KEY [ ( *scope* ) ]

|              |                                                                                  |
|--------------|----------------------------------------------------------------------------------|
| <b>KEY</b>   | Displays a list of KEYS.                                                         |
| <i>scope</i> | A symbol containing a FILE. If omitted, the list displays all KEYS in all FILEs. |

The **KEY** *type* in a #PROMPT statement indicates the prompt's *symbol* must contain the label of a KEY. A list of available keys from the data dictionary pops up when the #PROMPT is encountered on the Properties screen.

There may be a *scope* parameter that limits the KEYS available in the list. If *scope* names a FILE, the list displays all KEYS in the FILE. If there is no *scope* parameter, the list displays all KEYS in all FILEs.

Example:

```
#PROMPT('Which Key',KEY(%Primary)),%UseKey
```

## KEYCODE (list of keycodes)

---

### KEYCODE

The **KEYCODE** *type* in a #PROMPT statement indicates the prompt's *symbol* must contain a keycode or keycode equate label. A selection list of keycode equate labels from KEYCODES.EQU pops up when the user presses the ellipsis button next to the prompt on the Properties screen.

Example:

```
#PROMPT('Hot Key',KEYCODE),%ActiveKey
```

## OPENDIALOG (call Open File dialog)

---

### OPENDIALOG( *title, extensions* )

**OPENDIALOG** Calls the Windows common Open File dialog.

*title* A string constant containing the title to place on the file open dialog.

*extensions* A string constant containing the available file extension selections for the List Files of Type drop list.

The **OPENDIALOG** *type* in a #PROMPT statement indicates the prompt's *symbol* must contain the name of the file selected by the programmer from the Open File dialog (see FILEDIALOG in the *Language Reference*). The file must already exist on disk.

Example:

```
#PROMPT('Ask for File',OPENDIALOG('Pick File','Source|*.CLW')),%FileSymbol
```

## OPTFIELD (optional text or data field)

---

### OPTFIELD

The **OPTFIELD** *type* in a #PROMPT statement indicates the prompt's *symbol* may either contain text entered by the programmer or the label of a field in a data file. When a field is selected, the #PROMPT symbol contains the field name with an exclamation prepended (!FieldName). A list of available fields pops up when the programmer presses the "..." button next to the #PROMPT entry box on the Properties screen.

Example:

```
#PROMPT('Default Value',OPTFIELD),%DefaultValue
```

## OPTION (display radio buttons)

---

### OPTION

The **OPTION** *type* in a #PROMPT statement indicates the prompt's *symbol* receives the value of one of the *strings* in one of the following RADIO #PROMPT statements, unless the CHOICE attribute is present, then the *symbol* receives the ordinal position number of the RADIO #PROMPT the programmer chooses from an OPTION set instead of the *string* text. Each of the *strings* displays a radio button on the Properties screen when the #PROMPT is encountered.

Example:

```
#PROMPT('Ask for Choice',OPTION),%OptionSymbol
#PROMPT('Option One',RADIO)
#PROMPT('Option Two',RADIO)
#PROMPT('Option Three',RADIO)
```

## PICTURE (call picture formatter)

---

### PICTURE

The **PICTURE** *type* in a #PROMPT statement calls the picture formatter to create a picture token used to format data for display.

Example:

```
#PROMPT('Display Format',PICTURE),%DisplayPicture
```

## PROCEDURE (add to logical procedure tree)

---

### PROCEDURE

The **PROCEDURE** *type* in a #PROMPT statement indicates the value placed in the *symbol* is the name of a procedure in your application. This procedure name is added to the Application Generator's logical procedure call tree in the appropriate place.

Example:

```
#PROMPT('Next Procedure',PROCEDURE),%NextProcedure
```

## RADIO (one radio button)

### RADIO

The **RADIO** *type* in a #PROMPT statement creates one RADIO button for the closest preceding OPTION prompt. When selected, the RADIO's *string* is placed in the OPTION's *symbol* unless the CHOICE attribute or VALUE is present. With the CHOICE attribute on the OPTION, the *symbol* receives the ordinal position number of the RADIO #PROMPT the programmer chooses from an OPTION set instead of the *string* text. With the VALUE attribute on the RADIO, the *symbol* receives *value* text.

Example:

```
#PROMPT('Ask for Choice',OPTION),%OptionSymbol1
#PROMPT('Option One',RADIO)
#PROMPT('Option Two',RADIO)
#PROMPT('Option Three',RADIO)

#PROMPT('Ask for Another Choice',OPTION),%OptionSymbol2,CHOICE
#PROMPT('Option A',RADIO) #!%OptionSymbol2 receives 1
#PROMPT('Option B',RADIO) #!%OptionSymbol2 receives 2
#PROMPT('Option C',RADIO) #!%OptionSymbol2 receives 3

#PROMPT('Ask for Yet Another Choice',OPTION),%OptionSymbol3
#PROMPT('Option A',RADIO),VALUE('A') #!%OptionSymbol3 receives A
#PROMPT('Option B',RADIO),VALUE('B') #!%OptionSymbol3 receives B
#PROMPT('Option C',RADIO),VALUE('C') #!%OptionSymbol3 receives C
```

## SAVEDIALOG (call Save File dialog)

### SAVEDIALOG( *title*, *extensions* )

**SAVEDIALOG**     Calls the Windows common Save File dialog.

*title*             A string constant containing the title to place on the file save dialog.

*extensions*       A string constant containing the available file extension selections for the List Files of Type drop list.

The **SAVEDIALOG** *type* in a #PROMPT statement indicates the prompt's *symbol* must contain the name of the file selected by the programmer from the Save File dialog (see FILEDIALOG in the *Language Reference*). The file must not already exist on disk.

Example:

```
#PROMPT('Ask for FileName',SAVEDIALOG('Pick File','Source|*.CLW')),%FileSymbol1
```

## SPIN (spin box)

---

**SPIN**( *picture*, *low*, *high* [, *step* ] )

|                |                                                                                                                                                        |
|----------------|--------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>SPIN</b>    | Creates a spin control.                                                                                                                                |
| <i>picture</i> | A data entry picture token.                                                                                                                            |
| <i>low</i>     | A numeric constant or expression containing the lowest valid value.                                                                                    |
| <i>high</i>    | A numeric constant or expression containing the highest valid value.                                                                                   |
| <i>step</i>    | A numeric constant or expression containing the amount to change each increment between lowest and highest valid values. If omitted, the default is 1. |

The **SPIN** *type* in a #PROMPT statement creates a spin control for the programmer to select a valid number.

Example:

```
#PROMPT('How Many?',SPIN(@n2,1,10)),%SpinSymbol
```

## TEXT (text box)

---

**TEXT**

The **TEXT** *type* in a #PROMPT statement creates a multi-line text entry control for the programmer to enter text into.

Example:

```
#PROMPT('Comment?',TEXT),%Comments
```

## Display and Formatting Statements

### #BOXED (prompt group box)

```
#BOXED([string]) [, AT()] [, WHERE(expression)] [, CLEAR] [, HIDE] [, SECTION]
 prompts
#ENDBOXED
```

|                   |                                                                                                                                                                                                                                                                              |
|-------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>#BOXED</b>     | Creates a group box of <i>prompts</i> .                                                                                                                                                                                                                                      |
| <i>string</i>     | A string constant containing the text to display as the group box caption.                                                                                                                                                                                                   |
| <b>AT</b>         | Specifies the position of the group in the window, relative to the first prompt placed on the window from the Template (excluding the standard prompts on every procedure properties window). This attribute takes the same parameters as the Clarion language AT attribute. |
| <b>WHERE</b>      | Specifies the #BOXED is visible only for those instances where the <i>expression</i> is true.                                                                                                                                                                                |
| <i>expression</i> | An expression that specifies the condition for use.                                                                                                                                                                                                                          |
| <b>CLEAR</b>      | Specifies the <i>prompts</i> symbol values are cleared when disabled.                                                                                                                                                                                                        |
| <b>HIDE</b>       | Specifies the <i>prompts</i> are hidden if the WHERE <i>expression</i> is false when the dialog is first displayed. If no WHERE attribute is present, <i>prompts</i> are always hidden.                                                                                      |
| <b>SECTION</b>    | Specifies all AT() attributes for the <i>prompts</i> are positioned relative to the start of the #BOXED section.                                                                                                                                                             |
| <i>prompts</i>    | One or more #PROMPT statements. This may also contain #DISPLAY, #VALIDATE, #ENABLE, and #BUTTON statements.                                                                                                                                                                  |
| <b>#ENDBOXED</b>  | Terminates the group box of <i>prompts</i> .                                                                                                                                                                                                                                 |

The **#BOXED** statement creates a group box displaying the *string* as its caption. If the WHERE attribute is present, the *prompts* are hidden or visible based upon the evaluation of the *expression*. If the *expression* is true, the *prompts* are visible, otherwise they are hidden.

Example:

```
#PROMPT('Pick One',OPTION),%InputChoice #!These prompts on second page
#PROMPT('Choice One',RADIO)
#PROMPT('Choice Two',RADIO)
#BOXED('Choice Two Options'),WHERE(%InputChoice = 'Choice Two')
 #PROMPT('Screen Field',CONTROL),%SomeField
 #VALIDATE(%ScreenFieldType = 'LIST','Must select a list box')
#ENDBOXED
```

See Also: **#PROMPT, #VALIDATE**

## #DISPLAY (display-only prompt)

---

**#DISPLAY**( [ *string* ] ) [, AT( ) ]

|                 |                                                                                                                                                                                                                                                                                                            |
|-----------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>#DISPLAY</b> | Displays a string constant on a properties window.                                                                                                                                                                                                                                                         |
| <i>string</i>   | A string expression containing the text to display.                                                                                                                                                                                                                                                        |
| <b>AT</b>       | Specifies the size and position of the <i>string</i> display area in the window, allowing multiple lines of text. This attribute takes the same parameters as the Clarion language AT attribute. If a width and height are specified in the AT attribute, then the <i>string</i> will span multiple lines. |

The **#DISPLAY** statement displays the *string* on a properties window. If the *string* is omitted, a blank line is displayed. The display updates whenever the value in the *string* changes. **#DISPLAY** is not valid in a **#MODULE** section.

Example:

```
#DISPLAY() #!Display a blank line
#DISPLAY('Ask programmer to input some') #!Display a string
#PROMPT(' specific value',@s20),%InputSymbol
```

See Also: **#PROMPT**, **#GROUP**, **#BOXED**, **#ENABLE**, **#BUTTON**

## #IMAGE (display graphic)

---

**#IMAGE**( *picture* ) [, AT( ) ]

|                |                                                                                                                                                                  |
|----------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>#IMAGE</b>  | Displays a graphic image on a properties window.                                                                                                                 |
| <i>picture</i> | A string expression containing the name of the image file to display.                                                                                            |
| <b>AT</b>      | Specifies the size and position of the <i>picture</i> display area in the window. This attribute takes the same parameters as the Clarion language AT attribute. |

The **#IMAGE** statement displays the *picture* graphic image on a properties window. The display updates whenever the value in the *picture* changes. **#IMAGE** is not valid in a **#MODULE** section.

Example:

```
#IMAGE('SomePic.BMP') #!Display a bitmap
```

## #SHEET (declare a group of #TAB controls)

```
#SHEET [, HSCROLL] [, ADJUST]
 tabs
#ENDSHEET
```

|                  |                                                                                               |
|------------------|-----------------------------------------------------------------------------------------------|
| <b>#SHEET</b>    | Declares a group of #TAB controls.                                                            |
| <b>HSCROLL</b>   | Specifies scrolling #TAB controls.                                                            |
| <b>ADJUST</b>    | Specifies automatic prompt positioning if the #TAB controls create more than one row of tabs. |
| <i>tabs</i>      | Multiple #TAB control declarations.                                                           |
| <b>#ENDSHEET</b> | Terminates the group box of <i>prompts</i> .                                                  |

**#SHEET** declares a group of #TAB controls that offer the user multiple “pages” of prompts for the window. The multiple #TAB controls in the SHEET structure define the “pages” displayed to the user.

Example:

```
#UTILITY(ApplicationWizard,'Create a New Database Application'),WIZARD
#!
#SHEET
 #TAB('Application Wizard')
 #IMAGE('CMPAPP.BMP')
 #DISPLAY('This wizard will create a new Application. '),AT(90,8,235,24)
 #DISPLAY('To begin creating your new Application, click Next. '),AT(90)
 #ENDTAB
 #TAB('Application Wizard - File Usage'),FINISH(1)
 #IMAGE('WINAPP.BMP')
 #DISPLAY('You can gen procs for all files, or select them'),AT(90,8,235,24)
 #PROMPT('Use all files in DCT',CHECK),%GenAllFiles,AT(90,,180),DEFAULT(1)
 #ENDTAB
 #TAB('Select Files to Use'),WHERE(NOT %GenAllFiles),FINISH(1)
 #IMAGE('WINAPP.BMP')
 #PROMPT('File Select',FROM(%File)),%FileSelect,INLINE,SELECTION('File Select')
 #ENDTAB
 #TAB('Application Wizard - Finally...'),FINISH(1)
 #IMAGE('WINAPP.BMP')
 #DISPLAY('Old procs can be overwritten or new procs suppressed')
 #PROMPT('Overwrite existing procs',CHECK),%OverwriteAll,AT(90,,235),DEFAULT(0)
 #IMAGE('<255,1,4,127>'),AT(90,55)
 #DISPLAY('Your First Procedure is always overwritten!'),AT(125,54,200,20)
 #ENDTAB
#ENDSHEET
```

## #TAB (declare a page of a #SHEET control)

```
#TAB(text) [,FINISH(expression)] [,WHERE(expression)]
 prompts
#ENDTAB
```

|                   |                                                                                                                                                                      |
|-------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>#TAB</b>       | Declares a group of <i>prompts</i> that constitute one of the multiple “pages” within a #SHEET structure.                                                            |
| <i>text</i>       | A string constant containing the text to display on the tab, or as the title of the window, if the WIZARD attribute is present on the #UTILITY.                      |
| <b>FINISH</b>     | Specifies the “Finish” button is present. Valid only in a #UTILITY with the WIZARD attribute.                                                                        |
| <i>expression</i> | An expression that specifies whether the “Finish” button is enabled or disabled. If the expression is true, the button is enabled, if false, the button is disabled. |
| <b>WHERE</b>      | Specifies the #BOXED is visible only for those instances where the <i>expression</i> is true.                                                                        |
| <i>expression</i> | An expression that specifies the condition for use.                                                                                                                  |
| <i>prompts</i>    | One or more #PROMPT statements. This may also contain #DISPLAY, #VALIDATE, #ENABLE, and #BUTTON statements.                                                          |
| <b>#ENDTAB</b>    | Terminates the page of <i>prompts</i> .                                                                                                                              |

The **#TAB** structure declares a group of *prompts* that constitute one of the multiple “pages” of controls contained within a #SHEET structure. The multiple #TAB controls in the #SHEET structure define the “pages” displayed to the user.

Example:

```
#UTILITY(ApplicationWizard,'Create a New Database Application'),WIZARD
#!
#SHEET
 #TAB('Application Wizard'),FINISH(0) #!Finish button dim
 #IMAGE('CMPAPP.BMP')
 #DISPLAY('This wizard will create a new Application. '),AT(90,8,235,24)
 #DISPLAY('To begin creating your new Application, click Next. '),AT(90)
#ENDTAB
 #TAB('Application Wizard - File Usage'),FINISH(1) #!Finish button active
 #IMAGE('WINAPP.BMP')
 #DISPLAY('You can gen procs for all files, or select them'),AT(90,8,235,24)
 #PROMPT('Use all files in DCT',CHECK),%GenAllFiles,AT(90,,180),DEFAULT(1)
#ENDTAB
 #TAB('Select Files to Use'),WHERE(NOT %GenAllFiles),FINISH(1)
 #IMAGE('WINAPP.BMP')
 #PROMPT('File Select',FROM(%File)),%FileSelect,INLINE,SELECTION('File Select')
#ENDTAB
 #TAB('Application Wizard - Finally...'),FINISH(1)
 #IMAGE('WINAPP.BMP')
 #DISPLAY('Old procs can be overwritten or new procs suppressed')
 #PROMPT('Overwrite existing procs',CHECK),%OverwriteAll,AT(90,,235),DEFAULT(0)
 #IMAGE('<255,1,4,127>'),AT(90,55)
 #DISPLAY('Your First Procedure is always overwritten!'),AT(125,54,200,20)
#ENDTAB
#ENDSHEET
```



# 19 - SOURCE GENERATION CONTROL

## Template Logic Control Statements

### #FOR (generate code multiple times)

```
#FOR(symbol) [, WHERE(expression)] [, REVERSE]
 statements
#ENDFOR
```

|                   |                                                                                                                                                  |
|-------------------|--------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>#FOR</b>       | Loops through all instances of a multi-valued symbol.                                                                                            |
| <i>symbol</i>     | A multi-valued symbol.                                                                                                                           |
| <b>WHERE</b>      | Specifies the <i>statements</i> in the #FOR loop are executed only for those instances of the <i>symbol</i> where the <i>expression</i> is true. |
| <i>expression</i> | An expression that specifies the condition for execution.                                                                                        |
| <b>REVERSE</b>    | Specifies the #FOR loops through the instances of the <i>symbol</i> in reverse order.                                                            |
| <i>statements</i> | Target and/or Template Language statements.                                                                                                      |
| <b>#ENDFOR</b>    | Terminates the #FOR structure.                                                                                                                   |

**#FOR** is a loop structure which generates its *statements* once for each value contained in its *symbol* during source code generation. If there are no values in the *symbol*, no code is generated. #FOR must be terminated by **#ENDFOR**. If there is no #ENDFOR, an error message is issued during Template file pre-processing. A #FOR loop may be nested within another #FOR loop.

The #FOR loop begins with the first instance of the *symbol* (or last, if the **REVERSE** attribute is present) and processes through all instances of the *symbol*—it is not affected by any #FIX statements. If the **WHERE** attribute is present, the #FOR *statements* are executed only for those instances of the *symbol* where the *expression* is true. This creates a conditional #FOR loop.

Since #FOR is a loop structure, the #BREAK and #CYCLE statements may be used to control the loop. #BREAK immediately terminates #FOR loop processing and continues with the statement following the #ENDFOR that terminates the #FOR. #CYCLE immediately returns control to the #FOR statement to continue with the next instance of the *symbol*.

Example:

```
#FOR(%ScreenField),WHERE(%ScreenFieldType = 'LIST')
 #INSERT(%ListQueueBuild) #!Generate only for LIST controls
#ENDFOR
```

See Also:           **#BREAK, #CYCLE**

## #IF (conditionally generate code)

```
#IF(expression)
 statements
[#ELSIF(expression)
 statements]
[#ELSE
 statements]
#ENDIF
```

|                   |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|-------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>#IF</b>        | A conditional execution structure.                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| <i>expression</i> | Any Template Language expression which can evaluate to false (blank or zero) or true (any other value). The expression may contain Template symbols, constant values, and any of the arithmetic, Boolean, and logical operators documented in the <i>Language Reference</i> . Procedure calls are allowed. If the modulus division operator (%) is used in the expression, it must be delimited by at least one blank space on each side (to explicitly differentiate it from the Template symbols). |
| <i>statements</i> | One or more Clarion and/or Template Language statements.                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <b>#ELSIF</b>     | Provides an alternate <i>expression</i> to evaluate when preceding #IF and #ELSIF <i>expressions</i> are false.                                                                                                                                                                                                                                                                                                                                                                                      |
| <b>#ELSE</b>      | Provides alternate <i>statements</i> to execute when all preceding #IF and #ELSIF <i>expressions</i> are false.                                                                                                                                                                                                                                                                                                                                                                                      |
| <b>#ENDIF</b>     | Terminates the #IF structure.                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |

**#IF** selectively generates a group of *statements* depending on the evaluation of the *expression(s)*. The #IF structure consists of a #IF statement and all statements following it until the structure is terminated by **#ENDIF**. If there is no #ENDIF, an error message is issued during Template file pre-processing. #IF structures may be nested within other #IF structures.

**#ELSIF** and **#ELSE** are logical separators which separate the #IF structure into *statements* groups which are conditionally generated depending upon the evaluation of the *expression(s)*. There may be multiple #ELSIF statements within one #IF structure, but only one #ELSE.

When #IF is encountered during code generation:

- If the *expression* evaluates as true, only the *statements* following #IF are generated, up to the next following #ELSIF, #ELSE, or #ENDIF.
- If the *expression* evaluates as false, #ELSIF (if present) is evaluated in the same manner. If the #ELSIF *expression* is true, only the *statements* following it are generated, up to the following #ELSIF, #ELSE, or #ENDIF.

- If all preceding #IF and #ELSIF conditions evaluate false, only the *statements* following #ELSE (if present) are generated, up to the following #ENDIF. If there is no #ELSE, no code is generated.

Example:

```
#IF(SUB(%ReportControlStatement,1,6)='HEADER')
 #SET(%Indentation,%Indentation+1)
#ELSIF(SUB(%ReportControlStatement,1,6)='FOOTER')
 #SET(%Indentation,%Indentation+1)
#ELSIF(SUB(%ReportControlStatement,1,6)='DETAIL')
 #SET(%Indentation,%Indentation+1)
#ELSIF(SUB(%ReportControlStatement,1,6)='OPTION')
 #SET(%Indentation,%Indentation+1)
#ELSIF(SUB(%ReportControlStatement,1,5)='GROUP')
 #SET(%Indentation,%Indentation+1)
#ELSIF(SUB(%ReportControlStatement,1,5)='BREAK')
 #SET(%Indentation,%Indentation+1)
#ELSIF(SUB(%ReportControlStatement,1,4)='FORM')
 #SET(%Indentation,%Indentation+1)
#ENDIF
```

## #LOOP (iteratively generate code)

```
#LOOP [, | UNTIL(expression) |]
 | WHILE(expression) |
 | FOR(counter, start, end) [, BY(step)] |
 | TIMES(iterations) |
 statements
#ENDLOOP
```

|                   |                                                                                                                                                                                                                                       |
|-------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>#LOOP</b>      | Initiates an iterative statement execution structure.                                                                                                                                                                                 |
| <b>UNTIL</b>      | Evaluates its <i>expression</i> before each iteration of the #LOOP. If its <i>expression</i> evaluates to true, the #LOOP control sequence terminates.                                                                                |
| <i>expression</i> | Any Template language expression which can evaluate to false (blank or zero) or true (any other value).                                                                                                                               |
| <b>WHILE</b>      | Evaluates its <i>expression</i> before each iteration of the #LOOP. If its <i>expression</i> evaluates to false, the #LOOP control sequence terminates.                                                                               |
| <b>FOR</b>        | Initializes its <i>counter</i> to the <i>start</i> value, and increments it by the <i>step</i> value each time through the loop. When the <i>counter</i> is greater than the <i>end</i> value, the #LOOP control sequence terminates. |
| <i>counter</i>    | A user-defined symbol used as the loop counter.                                                                                                                                                                                       |
| <i>start</i>      | An expression containing the initial value to which to set the loop <i>counter</i> .                                                                                                                                                  |
| <i>end</i>        | An expression containing the ending value of the loop <i>counter</i> .                                                                                                                                                                |
| <b>BY</b>         | Explicitly defines the increment value for the <i>counter</i> .                                                                                                                                                                       |
| <i>step</i>       | An expression containing the increment value for the <i>counter</i> . If omitted, the <i>step</i> defaults to one (1).                                                                                                                |
| <b>TIMES</b>      | Loops the number of times specified by the <i>iterations</i> .                                                                                                                                                                        |
| <i>iterations</i> | An expression containing the number of times to loop.                                                                                                                                                                                 |
| <i>statements</i> | One or more target and/or Template Language statements.                                                                                                                                                                               |
| <b>#ENDLOOP</b>   | Terminates the #LOOP structure.                                                                                                                                                                                                       |

A **#LOOP** structure repetitively executes the *statements* within its structure. The #LOOP structure must be terminated by **#ENDLOOP**. If there is no #ENDLOOP, an error message is issued during Template file pre-processing. A #LOOP structure may be nested within another #LOOP structure.

The **#LOOP, UNTIL** or **#LOOP, WHILE** statements create exit conditions for the #LOOP. Their *expressions* are always evaluated at the top of the #LOOP, before the #LOOP is executed. A #LOOP WHILE structure

continuously loops as long as the *expression* is true. A `#LOOP UNTIL` structure continuously loops as long as the *expression* is false. The *expression* may contain Template symbols, constant values, and any of the arithmetic, Boolean, and logical operators documented in the *Language Reference*. Procedure calls are allowed. If the modulus division operator (%) is used in the *expression*, it must be followed by at least one blank space (to explicitly differentiate it from the Template symbols).

The `#LOOP, FOR` statement also creates an exit condition for the `#LOOP`. The `#LOOP` initializes the *counter* to the *start* value on its first iteration. The `#LOOP` automatically increments the *counter* by the *step* value on each subsequent iteration, then evaluates the *counter* against the *end* value. When the *counter* is greater than the *end*, the `#LOOP` control sequence terminates.

`#LOOP` (without `WHILE`, `UNTIL`, or `FOR`) loops continuously, unless a `#BREAK` or `#RETURN` statement is executed. `#BREAK` terminates the `#LOOP` and continues execution with the statement following the `#LOOP` structure. All *statements* within a `#LOOP` structure are executed unless a `#CYCLE` statement is executed. `#CYCLE` immediately gives control back to the top of the `#LOOP` for the next iteration, without executing any statements following the `#CYCLE` in the `#LOOP`.

Example:

```
#SET(%LoopBreakFlag, 'NO')
#LOOP #!Continuous loop
 #INSERT(%SomeRepeatedCodeGroup)
 #IF(%LoopBreakFlag = 'YES') #!Check break condition
 #BREAK
 #ENDIF
#ENDLOOP

#SET(%LoopBreakFlag, 'NO')
#LOOP, UNTIL(%LoopBreakFlag = 'YES') #!Loop until condition is true
 #INSERT(%SomeRepeatedCodeGroup)
#ENDLOOP

#SET(%LoopBreakFlag, 'NO')
#LOOP, WHILE(%LoopBreakFlag = 'NO') #!Loop while condition is true
 #INSERT(%SomeRepeatedCodeGroup)
#ENDLOOP
```

See Also: `#BREAK`, `#CYCLE`

## #BREAK (break out of a loop)

---

### #BREAK

The **#BREAK** statement immediately breaks out of the **#FOR** or **#LOOP** structure in which it is enclosed. Control passes to the next statement following the **#ENDFOR** or **#ENDLOOP**. **#BREAK** is only valid within a **#FOR** or **#LOOP** structure, else an error is generated during Template file pre-processing. **#BREAK** acts as a **#RETURN** statement if issued from within a **#GROUP** inserted in the loop (unless it is within a **#FOR** or **#LOOP** structure completely contained within the **#GROUP**).

Example:

```
#SET(%StopFile,'CUSTOMER')
#FOR(%File)
 #IF (UPPER(%File) = %StopFile)
 #BREAK
 #ENDIF
 OPEN(%File)
#ENDFOR
```

## #CYCLE (cycle to top of loop)

---

### #CYCLE

The **#CYCLE** statement immediately passes control back to the top of the **#FOR** or **#LOOP** structure in which it is enclosed to begin the next iteration. **#CYCLE** is only valid within a **#FOR** or **#LOOP** structure, else an error is generated during Template file pre-processing. **#CYCLE** acts as a **#RETURN** statement if issued from within a **#GROUP** inserted in the loop (unless it is within a **#FOR** or **#LOOP** structure completely contained within the **#GROUP**).

Example:

```
#SET(%StopFile,'CUSTOMER')
#FOR(%File)
 #IF (UPPER(%File) <> %StopFile)
 OPEN(%File)
 #CYCLE
 #ELSE
 #BREAK
 #ENDIF
#ENDFOR
```

## #CASE (conditional execution structure)

```
#CASE(condition)
#OF(expression)
[#OROF(expression)]
 statements
[#ELSE
 statements]
#ENDCASE
```

|                   |                                                                                                                                                                                                                  |
|-------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>#CASE</b>      | Initiates a selective execution structure.                                                                                                                                                                       |
| <i>condition</i>  | Any Template Language expression which returns a value.                                                                                                                                                          |
| <b>#OF</b>        | The #OF <i>statements</i> are executed when the #OF <i>expression</i> is equal to the <i>condition</i> of the CASE. There may be many #OF options in a #CASE structure.                                          |
| <i>expression</i> | Any Template Language expression which returns a value.                                                                                                                                                          |
| <b>#OROF</b>      | The #OROF <i>statements</i> are executed when the #OROF <i>expression</i> is equal to the <i>condition</i> of the #CASE. There may be many #OROF options associated with one #OF option.                         |
| <b>#ELSE</b>      | The #ELSE <i>statements</i> are executed when all preceding #OF and #OROF <i>expressions</i> are not equal to the <i>condition</i> of the #CASE. #ELSE (if used) must be the last option in the #CASE structure. |
| <i>statements</i> | Any valid executable source code.                                                                                                                                                                                |
| <b>#ENDCASE</b>   | Terminates the #CASE structure.                                                                                                                                                                                  |

A #CASE structure selectively executes *statements* based on equivalence between the #CASE *condition* and one of the #OF or #OROF *expressions*. If there is no exact match, the *statements* following #ELSE are executed. The #CASE structure must be terminated by #ENDCASE. If there is no #ENDCASE, an error message is issued during Template file pre-processing. #CASE structures may be nested within other #CASE structures.

Example:

```
#CASE(%ScreenField)
#OF('?Ok')
 #INSERT(%OkButtonGroup)
#OF('?Cancel')
#OROF('?Exit')
 #INSERT(%CancelButtonGroup)
#ELSE
 #INSERT(%OtherControlsGroup)
#ENDCASE
```

## #INDENT (change indentation level)

---

### #INDENT( *value* )

|                |                                                                                                                                                 |
|----------------|-------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>#INDENT</b> | Changes the indentation level of generated code.                                                                                                |
| <i>value</i>   | An expression that resolves to a positive or negative integer which specifies the amount and direction to change the current indentation level. |

The **#INDENT** statement changes the indentation level of generated code by the amount and direction specified by the *value* parameter. If the *value* is positive, the indentation level increases (moves right). If the *value* is negative, the indentation level decreases (moves left).

Example:

```
#INDENT(-2)
```

```
#!Change indent level left 2 spaces
```

See Also:

**#INSERT**

## #INSERT (insert code from a #GROUP)

```
#INSERT(symbol [(set)] [, parameters] [, returnvalue] [, NOINDENT]
```

|                    |                                                                                                                                                                                                  |
|--------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>#INSERT</b>     | Inserts code from a #GROUP.                                                                                                                                                                      |
| <i>symbol</i>      | A symbol that names a #GROUP section.                                                                                                                                                            |
| <i>set</i>         | The #TEMPLATE <i>name</i> parameter for the template set to which the #GROUP belongs. If omitted, the #GROUP must be of the same template set <i>name</i> as the #PROCEDURE in which it is used. |
| <i>parameters</i>  | The parameters passed to the #GROUP. Each parameter must be separated by a comma. All parameters defined for the #GROUP must be passed; they may not be omitted.                                 |
| <i>returnvalue</i> | A symbol to receive the value returned by the #RETURN statement.                                                                                                                                 |
| <b>NOINDENT</b>    | Specifies the code inserted retains the indentation in its #GROUP—any source in column 1 in the #GROUP is generated into column 1, no matter where the #INSERT is placed.                        |

The **#INSERT** statement places, at the exact position the #INSERT is located within the Template code, the code from the #GROUP named by the *symbol*. The *set* parameter specifies the #TEMPLATE that contains the #GROUP. This allows any Template to use #GROUP code from any other registered Template.

The *parameters* passed to the #GROUP fall into two categories: value-parameters and variable-parameters. Value-parameters are declared by the #GROUP as a user-defined symbol, while variable-parameters are declared by the #GROUP as a user-defined symbol with a prepended asterisk (\*). Either a symbol or an expression may be passed as a value-parameter. Only a symbol may be passed as a variable-parameter.

The *returnvalue* symbol receives the value returned by the #GROUP from the #RETURN statement that terminates the #GROUP. If the #GROUP does not contain a #RETURN statement, or that #RETURN does not have a parameter, then the value received is an empty string ('').

Example:

```
#INSERT(%SomeGroup) #!Ordinary insert
#INSERT(%GenerateFormulas(Clarion)) #!Insert #GROUP from Clarion Template
#INSERT(%FileRecordFilter,%Secondary) #!Insert #GROUP with passed parameter
#INSERT(%FileRecordFilter(Clarion),%Primary,%Secondary)
 #!#GROUP from Clarion Template with two passed parameters
```

See Also:

#GROUP, #CALL, #RETURN

## #CALL (insert code from a #GROUP, without indention)

```
#CALL(symbol [(set)] [, parameters]) [, returnvalue]
```

|                    |                                                                                                                                                                                                  |
|--------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>#CALL</b>       | Inserts code from a #GROUP, retaining the indention in its #GROUP.                                                                                                                               |
| <i>symbol</i>      | A symbol that names a #GROUP section.                                                                                                                                                            |
| <i>set</i>         | The #TEMPLATE <i>name</i> parameter for the template set to which the #GROUP belongs. If omitted, the #GROUP must be of the same template set <i>name</i> as the #PROCEDURE in which it is used. |
| <i>parameters</i>  | The parameters passed to the #GROUP. Each parameter must be separated by a comma. All parameters defined for the #GROUP must be passed; they may not be omitted.                                 |
| <i>returnvalue</i> | A symbol to receive the value returned by the #RETURN statement.                                                                                                                                 |

The **#CALL** statement places the code from the #GROUP named by the *symbol*, retaining the indention in its #GROUP—any source in column 1 in the #GROUP is generated into column 1, no matter where the #CALL is placed. This is equivalent to using #INSERT with the NOINDENT parameter.

The *set* parameter specifies the #TEMPLATE that contains the #GROUP. This allows any Template to use #GROUP code from any other registered Template. The *parameters* passed to the #GROUP fall into two categories: value-parameters and variable-parameters. Value-parameters are declared by the #GROUP as a user-defined symbol, while variable-parameters are declared by the #GROUP as a user-defined symbol with a prepended asterisk (\*). Either a symbol or an expression may be passed as a value-parameter. Only a symbol may be passed as a variable-parameter.

The *returnvalue* symbol receives the value returned by the #GROUP from the #RETURN statement that terminates the #GROUP. If the #GROUP does not contain a #RETURN statement, or that #RETURN does not have a parameter, then the value received is an empty string ('').

Example:

```
#CALL(%SomeGroup) #!Ordinary insert retaining original indention
#CALL(%GenerateFormulas(Clarion)) #!Insert #GROUP from Clarion Template
#CALL(%FileRecordFilter,%Secondary) #!Insert #GROUP with passed parameter
#CALL(%FileRecordFilter(Clarion),%Primary,%Secondary)
 #!#GROUP from Clarion Template with two passed parameters
```

See Also:

#GROUP, #INSERT, #INVOKE, #RETURN

## #INVOKE (insert code from a named #GROUP)

```
#INVOKE(symbol [, parameters]) [, returnvalue] [, NOINDENT]
```

|                    |                                                                                                                                                                           |
|--------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>#INVOKE</b>     | Inserts code from a named #GROUP.                                                                                                                                         |
| <i>symbol</i>      | A symbol containing the name of a #GROUP section (including the #TEMPLATE to which it belongs).                                                                           |
| <i>parameters</i>  | The parameters passed to the #GROUP. Each parameter must be separated by a comma. All parameters defined for the #GROUP must be passed; they may not be omitted.          |
| <i>returnvalue</i> | A symbol to receive the value returned by the #RETURN statement.                                                                                                          |
| <b>NOINDENT</b>    | Specifies the code inserted retains the indentation in its #GROUP—any source in column 1 in the #GROUP is generated into column 1, no matter where the #INSERT is placed. |

The **#INVOKE** statement places, at the exact position the **#INVOKE** is located within the Template code, the code from the #GROUP named in the *symbol*. The *symbol* must contain the name of a #GROUP, including the #TEMPLATE set to which it belongs. The main difference between **#INVOKE** and **#CALL** is the *symbol* parameter, which is a variable containing the #GROUP to call in **#INVOKE** and a constant naming the #GROUP to call in **#CALL**.

The *parameters* passed to the #GROUP fall into two categories: value-parameters and variable-parameters. Value-parameters are declared by the #GROUP as a user-defined symbol, while variable-parameters are declared by the #GROUP as a user-defined symbol with a prepended asterisk (\*). Either a symbol or an expression may be passed as a value-parameter. Only a symbol may be passed as a variable-parameter.

The *returnvalue* symbol receives the value returned by the #GROUP from the #RETURN statement that terminates the #GROUP. If the #GROUP does not contain a #RETURN statement, or that #RETURN does not have a parameter, then the value received is an empty string ('').

Example:

```
#SET(%SomeGroup,%GenerateFormulas(Clarion')) #!Set variable to a #GROUP name
#INVOKE(%SomeGroup)

#SET(%SomeGroup,%FileRecordFilter(Clarion')) #!Set variable to another #GROUP name
#INVOKE(%SomeGroup,%Secondary) #!Insert #GROUP with passed parameter
#INVOKE(%SomeGroup,%Primary,%Secondary) #!#GROUP with two passed parameters
```

See Also:

**#GROUP, #INSERT, #CALL, #RETURN**

## #RETURN (return from #GROUP)

---

**#RETURN**( [ *returnvalue* ] )

|                    |                                                                                                                      |
|--------------------|----------------------------------------------------------------------------------------------------------------------|
| <b>#RETURN</b>     | Immediately returns from the #GROUP.                                                                                 |
| <i>returnvalue</i> | An expression containing the value to return to the calling statement. If omitted, an empty string (``) is returned. |

The **#RETURN** statement immediately returns control to the statement following the **#INSERT**, **#CALL**, or **#INVOKE** that called the **#GROUP**, or returns the *returnvalue* to the **CALL** or **INVOKE** built-in procedures. If the **#INSERT**, **#CALL**, or **#INVOKE** that called the **#GROUP** are set to receive the *returnvalue*, the value is placed in the symbol named in the **#INSERT**, **#CALL**, or **#INVOKE** statement.

**#RETURN** is only valid in a **#GROUP** section.

Example:

```
#GROUP(%ProcessListGroup,%PassedControl)
#FIX(%ScreenField,%PassedControl)
#IF (%ScreenFieldType <> 'LIST')
 #UNFIX(%ScreenField)
#RETURN
#ENDIF
```

See Also:

**#GROUP**, **#INSERT**, **#CALL**, **CALL**, **#INVOKE**, **INVOKE**



## #OPEN (open source file)

**#OPEN( *file* ) [, READ ]**

|              |                                                                                                                                    |
|--------------|------------------------------------------------------------------------------------------------------------------------------------|
| <b>#OPEN</b> | Opens a disk file to receive generated source code.                                                                                |
| <i>file</i>  | A string constant, template symbol, or expression containing a DOS file specification. This may be a fully qualified DOS pathname. |
| <b>READ</b>  | Opens the file as read-only. The file cannot be already open for output.                                                           |

The **#OPEN** statement opens a disk file to receive the source code generated by **#GENERATE**. If the *file* does not exist, it is created. If the *file* already exists, it is opened in “append source” mode. If the *file* is already open, a source generation error is produced. The *file* is automatically selected as the active source output destination.

If the **READ** attribute is present, the file is opened in read-only mode so the **#READ** statement can read it as an ASCII text file. Only one file can be open for input at one time.

Example:

```
#SET(%ProgramFile,(%Application & '.$$$')) #!Temp program filename
#OPEN(%ProgramFile) #!Open existing program file
#GENERATE(%Program) #!Generate main program header
#CLOSE(%ProgramFile) #!Close output file

#OPEN(%ProgramFile),READ #!Open it in read-only mode
#DECLARE(%ASCIIFileRecord)
#LOOP
 #READ(%ASCIIFileRecord)
 #! Parse the line and do something with it
 #IF(%ASCIIFileRecord = %EOF)
 #BREAK
 #ENDIF
#ENDLOOP
#CLOSE(%ProgramFile),READ
```

See Also:           **#READ, #CLOSE**

## #CLOSE (close source file)

**#CLOSE( [*file*] [, READ ]**

**#CLOSE**

Closes an open generated source code disk file.

*file*

A string constant, template symbol, or expression containing a DOS file specification. This may be a fully qualified DOS pathname. If omitted, the current disk file receiving generated source code is closed.

**READ**

Closes the read-only input file.

The **#CLOSE** statement closes an open disk file receiving the generated source code. If the *file* is omitted, the current disk file receiving generated source code is closed. If the *file* does not exist, or is already closed, a source generation error is produced.

Example:

```
#SET(%NewProgramFile,(%Application & '.$$$')) #!Temp new program filename
#CREATE(%NewProgramFile) #!Create new program file
#GENERATE(%Program) #!Generate main program header
#CLOSE(%NewProgramFile) #!Create new program file

#OPEN(%ProgramFile),READ #!Open it in read-only mode
#DECLARE(%ASCIIFileRecord)
#LOOP
 #READ(%ASCIIFileRecord)
 #! Parse the line and do something with it
 #IF(%ASCIIFileRecord = %EOF)
 #BREAK
 #ENDIF
#ENDLOOP
#CLOSE(%ProgramFile),READ #!Close the read-only file
```

See Also:

**#OPEN, #READ**

## #READ (read one line of a source file)

---

### #READ( *symbol* )

**#READ** Reads the next record from the opened read-only file.  
*symbol* The symbol to receive the text from the file.

The **#READ** statement reads the next record (up to the next CR/LF encountered) from open read-only file. The *symbol* receives the text from the file. If the last record has been read, the *symbol* will contain a value equivalent to the %EOF built-in symbol.

Example:

```
#OPEN(%ProgramFile),READ #!Open it in read-only mode
#DECLARE(%ASCIIFileRecord)
#LOOP
 #READ(%ASCIIFileRecord)
 #! Parse the line and do something with it
 #IF(%ASCIIFileRecord = %EOF)
 #BREAK
 #ENDIF
#ENDLOOP
#CLOSE(%ProgramFile),READ #!Close the read-only file
```

See Also:           **#OPEN, #CLOSE**



## #APPEND (add to source file)

**#APPEND( *file* ) [ , SECTION ]**

|                |                                                                                                                                    |
|----------------|------------------------------------------------------------------------------------------------------------------------------------|
| <b>#APPEND</b> | Adds the <i>file</i> contents to the end of the current source output destination file.                                            |
| <i>file</i>    | A string constant, template symbol, or expression containing a DOS file specification. This may be a fully qualified DOS pathname. |
| <b>SECTION</b> | Specifies the error position of embedded source code is correctly patched to reflect the actual position in the generated code.    |

The **#APPEND** statement adds the complete contents of the *file* to the end of the current source output destination file. The contents of the *file* are NOT interpreted for source generation purposes. Therefore, the *file* should not contain any Template Language code.

If the *file* does not exist, **#APPEND** is ignored and source generation continues.

Example:

```
#FOR(%Module)
 #SET(%TempModuleFile,(%Module & '.$$$')) #!Set temp module file
 #CREATE(%TempModuleFile) #!Create temp module file
 #FOR(%ModuleProcedure) #!For all procs in module
 #FIX(%Procedure,%ModuleProcedure) #!Fix current procedure
 #GENERATE(%Procedure) #!Generate procedure code
 #ENDFOR #!EndFor all procs in module
 #SET(%ModuleFile,(%Module & '.CLW')) #!Set to current module file
 #CREATE(%ModuleFile) #!Create module file
 #GENERATE(%Module) #!Generate module header
 #APPEND(%TempModuleFile),SECTION #!Add generated procedures
#ENDFOR
```

See Also:           **#SECTION**

## #SECTION (define code section)

---

```
#SECTION
 statements
#ENDSECTION
```

**#SECTION** Marks the beginning of a section of generated code.  
*statements* Any template or target language statements.  
**#ENDSECTION** Terminates the **#SECTION**.

The **#SECTION** structure defines a contiguous section of generated code so that the position of embedded source code is correctly patched up for error positions in the resulting generated code. **#APPEND** with the **SECTION** attribute performs the patch.

Example:

```
#CREATE(%Temp) #!Create temp module file
#SECTION
#EMBED(%MyEmbed,@s30)
#ENDSECTION
#CLOSE
#CREATE('target.clw')
 CODE
#APPEND(%Temp),SECTION #!Correctly patch position of #EMBED
#CLOSE
```

See Also: **#APPEND**

## #REMOVE (delete a source file)

### #REMOVE( *file* )

|                |                                                                                                                                    |
|----------------|------------------------------------------------------------------------------------------------------------------------------------|
| <b>#REMOVE</b> | Deletes a source output file.                                                                                                      |
| <i>file</i>    | A string constant, template symbol, or expression containing a DOS file specification. This may be a fully qualified DOS pathname. |

The **#REMOVE** statement deletes the specified source output *file*. If the *file* does not exist, **#REMOVE** is ignored and source generation continues.

Example:

```
#FOR(%Module)
 #SET(%TempModuleFile,(%Module & '.$$$'))
 #CREATE(%TempModuleFile)
 #FOR(%ModuleProcedure)
 #FIX(%Procedure,%ModuleProcedure)
 #GENERATE(%Procedure)
 #ENDFOR
 #SET(%ModuleFile,(%Module & '.CLW'))
 #CREATE(%ModuleFile)
 #GENERATE(%Module)
 #APPEND(%TempModuleFile)
 #REMOVE(%TempModuleFile)
#ENDFOR
 #!Set temp module file
 #!Create temp module file
 #!For all procs in module
 #!Fix current procedure
 #!Generate procedure code
 #!EndFor all procs in module
 #!Set to current module file
 #!Create module file
 #!Generate module header
 #!Add generated procedures
 #!Delete the temporary file
```

## #REPLACE (conditionally replace source file)

**#REPLACE**( *oldfile*, *newfile* )

|                 |                                                                                                                                    |
|-----------------|------------------------------------------------------------------------------------------------------------------------------------|
| <b>#REPLACE</b> | Performs “intelligent” file replacement.                                                                                           |
| <i>oldfile</i>  | A string constant, template symbol, or expression containing a DOS file specification. This may be a fully qualified DOS pathname. |
| <i>newfile</i>  | A string constant, template symbol, or expression containing a DOS file specification. This may be a fully qualified DOS pathname. |

The **#REPLACE** statement performs a binary comparison between the contents of the *oldfile* and *newfile*. If the contents of the *oldfile* are different from the contents of the *newfile* (or the *oldfile* does not exist), then the *oldfile* is deleted and the *newfile* is renamed to the *oldfile*. If the two files are identical, then no action is taken. If the *newfile* does not exist, **#REPLACE** is ignored and source generation continues.

Example:

```
#FOR(%Module)
 #SET(%TempModuleFile,(%Module & '.$$$')) #!Set temp module file
 #CREATE(%TempModuleFile) #!Create temp module file
 #GENERATE(%Module) #!Generate module header
 #FOR(%ModuleProcedure) #!For all procs in module
 #FIX(%Procedure,%ModuleProcedure) #!Fix current procedure
 #GENERATE(%Procedure) #!Generate procedure code
 #ENDFOR #!EndFor all procs in module
 #SET(%ModuleFile,(%Module & '.CLW')) #!Set to existing module file
 #REPLACE(%ModuleFile,%TempModuleFile) #!Replace old with new (if changed)
#ENDFOR
```

## #PRINT (print a source file)

---

### #PRINT( *file*, *title* )

|               |                                                                                                                                    |
|---------------|------------------------------------------------------------------------------------------------------------------------------------|
| <b>#PRINT</b> | Prints a <i>file</i> to the current Windows printer.                                                                               |
| <i>file</i>   | A string constant, template symbol, or expression containing a DOS file specification. This may be a fully qualified DOS pathname. |
| <i>title</i>  | A string constant, template symbol, or expression containing the title to generate for the <i>file</i> .                           |

The **#PRINT** statement prints the contents of the *file* to the Windows default printer.

Example:

```
#FOR(%Module)
 #SET(%ModuleFile,(%Module & '.CLW')) #!Set to existing module file
 #PRINT(%ModuleFile,"Printout ' & %ModuleFile)
#ENDFOR
```

## Source Generation Statements

### #GENERATE (generate source code section)

#### #GENERATE( *section* )

**#GENERATE** Generates a section of the application.  
*section* One of the following built-in symbols: %Program, %Module, or %Procedure. This symbol indicates the portion of the application to generate.

The **#GENERATE** statement generates the source code for the specified *section* of the application by executing the Template Language statements contained within that *section*. **#GENERATE** should only be used within the **#APPLICATION** or a **#UTILITY** section of the Template.

When *section* is:

|            |                                                                                                             |
|------------|-------------------------------------------------------------------------------------------------------------|
| %Program   | The <b>#PROGRAM</b> section of the Template is generated.                                                   |
| %Module    | The appropriate <b>#MODULE</b> section of the Template is generated.                                        |
| %Procedure | The appropriate <b>#PROCEDURE</b> section of the Template for the current value of %Procedure is generated. |

Example:

```
#GENERATE(%Program) #!Generate program header
#FOR(%Module) #!
 #GENERATE(%Module) #!Generate module header
 #FOR(%ModuleProcedure) #!For all procs in module
 #FIX(%Procedure,%ModuleProcedure) #!Fix current procedure
 #GENERATE(%Procedure) #!Generate procedure code
 #ENDFOR #!EndFor all procs in module
#ENDFOR #!EndFor all modules
```

### #ABORT (abort source generation)

#### #ABORT

The **#ABORT** statement immediately terminates source generation by the previous **#GENERATE** statement. **#ABORT** may be placed in any template section.

Example:

```
#IF(%ValidRangeKey=%Null)
 #ERROR(%Procedure & ' Range Error: The range field is not in the primary key!')
 #ABORT
#ENDIF
```

See Also: **#GENERATE**

## #SUSPEND (begin conditional source)

### #SUSPEND( *name* )

|                 |                                                                 |
|-----------------|-----------------------------------------------------------------|
| <b>#SUSPEND</b> | Marks the start of a section of conditionally generated source. |
| <i>name</i>     | The name of the #SUSPEND section for later use in #QUERY.       |

The **#SUSPEND** statement marks the start of a section of source that is generated only if a #RELEASE statement is encountered or an implicit #RELEASE occurs. This allows empty unnecessary “boiler-plate” code to be easily removed from the generated source. The end of the #SUSPEND section must be delimited by a matching #RESUME statement.

These #SUSPEND sections may be nested within each other to as many levels as necessary. A #RELEASE encountered in an inner nested section commits source generation for all the outer nested levels in which it is contained, also. A #EMBED that contains source to generate performs an implied #RELEASE. Any generated source output also performs an implied #RELEASE. Therefore, an explicit #RELEASE statement is not always necessary. The #? statement defines an individual conditional source line that does not perform the implied #RELEASE.

Example:

```
ACCEPT
#SUSPEND #!Begin suspended generation
 #?CASE SELECTED()
 #FOR(%ScreenField)
 #SUSPEND
 #?OF %ScreenField
 #EMBED(%ScreenSetups,'Control selected code'),%ScreenField
 #!Implied #RELEASE from the #EMBED of both nested sections
 #RESUME
 END
#RESUME #!End suspended generation
#SUSPEND #!Begin suspended generation
 #?CASE EVENT()
 #SUSPEND
 #?OF EVENT:AlertKey
 #SUSPEND
 #?CASE KEYCODE()
 #FOR %HotKey
 #RELEASE #!Explicit #RELEASE
 #?OF %HotKey
 #EMBED(%HotKeyProc,'Hot Key code'),%HotKey
 #ENDFOR
 #?END
 #RESUME
#?END
#RESUME #!End suspended generation
END
```

See Also: **#RELEASE, #RESUME, #?, #QUERY**

## #RELEASE (commit conditional source generation)

### #RELEASE

The **#RELEASE** enables source generation in a **#SUSPEND** section. This allows empty unnecessary “boiler-plate” code to be easily removed from the generated source. The code in a **#SUSPEND** section is generated only when a **#RELEASE** statement is encountered.

**#SUSPEND** sections may be nested within each other to as many levels as necessary. A **#RELEASE** encountered in an inner nested section commits source generation for all the outer nested levels in which it is contained, also.

A **#EMBED** that contains source to generate performs an implied **#RELEASE**. Any generated source output also performs an implied **#RELEASE**. Therefore, an explicit **#RELEASE** statement is not always necessary. The **#!?** statement defines an individual conditional source line that does not perform the implied **#RELEASE**.

Example:

```
ACCEPT
#SUSPEND #!Begin suspended generation
 #?CASE SELECTED()
 #FOR(%ScreenField)
 #SUSPEND
 #?OF %ScreenField
 #EMBED(%ScreenSetups,'Control selected code'),%ScreenField
 #!Implied #RELEASE from the #EMBED of both nested sections
 #RESUME
 #?END
#RESUME #!End suspended generation

#SUSPEND #!Begin suspended generation
 #?CASE EVENT()
 #SUSPEND
 #?OF EVENT:AlertKey
 #SUSPEND
 #?CASE KEYCODE()
 #FOR %HotKey
 #RELEASE #!Explicit #RELEASE
 #?OF %HotKey
 #EMBED(%HotKeyProc,'Hot Key code'),%HotKey
 #ENDFOR
 #?END
 #RESUME
#?END
#RESUME #!End suspended generation
END
```

See Also: **#SUSPEND, #RESUME, #?**

## #RESUME (delimit conditional source)

### #RESUME

The **#RESUME** statement marks the end of a section of source that is generated only if a **#RELEASE** statement is encountered. This allows empty unnecessary “boiler-plate” code to be easily removed from the generated source. The beginning of the section must be delimited by a matching **#SUSPEND** statement.

These **#SUSPEND** sections may be nested within each other to as many levels as necessary. A **#RELEASE** encountered in an inner nested section commits source generation for all the outer nested levels in which it is contained, also.

A **#EMBED** that contains source to generate performs an implied **#RELEASE**. Any generated source output also performs an implied **#RELEASE**. Therefore, an explicit **#RELEASE** statement is not always necessary. The **#!** statement defines an individual conditional source line that does not perform the implied **#RELEASE**.

When a **#RESUME** is executed without the output to the file being released, any conditional lines of code are un-done back to the matching **#SUSPEND**.

Example:

```
ACCEPT
#SUSPEND #!Begin suspended generation
 #?CASE SELECTED()
 #FOR(%ScreenField)
 #SUSPEND
 #?OF %ScreenField
 #EMBED(%ScreenSetups,'Control selected code'),%ScreenField
 #!Implied #RELEASE from the #EMBED of both nested sections
 #RESUME
 #?END
#RESUME #!End suspended generation
#SUSPEND #!Begin suspended generation
 #?CASE EVENT()
 #SUSPEND
 #?OF EVENT:AlertKey
 #SUSPEND
 #?CASE KEYCODE()
 #FOR %HotKey
 #RELEASE #!Explicit #RELEASE
 #?OF %HotKey
 #EMBED(%HotKeyProc,'Hot Key code'),%HotKey
 #ENDFOR
 #?END
 #RESUME
 #?END
#RESUME #!End suspended generation
END
```

See Also: **#SUSPEND, #RELEASE, #?**

## #? (conditional source line)

### #?statement

|                  |                                                                                                         |
|------------------|---------------------------------------------------------------------------------------------------------|
| <b>#?</b>        | Defines a single line of source code generated only if #RELEASE commits the conditional source section. |
| <i>statement</i> | A single line of target language code. This may contain template symbols.                               |

The #? statement defines a single line of source code that is generated only if a #RELEASE statement is encountered. This allows empty unnecessary “boiler-plate” code to be easily removed from the generated source.

A #EMBED that contains source to generate performs an implied #RELEASE. Any generated source output also performs an implied #RELEASE. Therefore, an explicit #RELEASE statement is not always necessary. The #? statement defines an individual conditional source line that does not perform the implied #RELEASE. When a #RESUME is executed without the output to the file being released, any conditional lines of code are un-done back to the matching #SUSPEND.

Example:

```
ACCEPT #!Unconditional source line
#SUSPEND
 #?CASE SELECTED() #!Conditional source line
 #FOR(%ScreenField)
 #SUSPEND
 #?OF %ScreenField #!Conditional source line
 #EMBED(%ScreenSetups,'Control selected code'),%ScreenField
 #RESUME
 #?END #!Conditional source line
#RESUME
#SUSPEND
 #?CASE EVENT() #!Conditional source line
 #SUSPEND
 #?OF EVENT:AlertKey #!Conditional source line
 #SUSPEND
 #?CASE KEYCODE() #!Conditional source line
 #FOR %HotKey
 #RELEASE
 #?OF %HotKey #!Conditional source line
 #EMBED(%HotKeyProc,'Hot Key code'),%HotKey
 #ENDFOR
 #?END #!Conditional source line
 #RESUME
#?END #!Conditional source line
#RESUME
END #!Unconditional source line
```

See Also: **#SUSPEND, #RELEASE, #RESUME**

## #QUERY (conditionally generate source)

**#QUERY**( *section*, *source* )

|                |                                                                                 |
|----------------|---------------------------------------------------------------------------------|
| <b>#QUERY</b>  | Generates <i>source</i> if the named #SUSPEND <i>section</i> has been released. |
| <i>section</i> | The name of the #SUSPEND section to #QUERY.                                     |
| <i>source</i>  | The code to generate.                                                           |

The **#QUERY** statement conditionally generates its *source* based on whether the named #SUSPEND *section* has been released (either explicitly or implicitly). If the #SUSPEND *section* has been released then #QUERY generates its source, without affecting the release of any other nested #SUSPEND section it may be in—it does not create an implicit #RELEASE.

Example:

```
#SUSPEND(Fred) #!Begin suspended generation section named "Fred"
#?1
#SUSPEND #!Begin unnamed suspended generation section
#?2
#QUERY(Fred,'3') #!Generate "3" only if the "Fred" section is released
#?4
#RESUME
5 #!Unconditional generation causes implicit #RELEASE of "Fred"
#RESUME #!End "Fred" section

#!The above code will generate 1, 3, 5 on successive output lines
#!The 2 and 4 don't generate because the unnamed section was not released
```

See Also:           **#SUSPEND**

## External Code Execution Statements

### #RUN (execute program)

---

**#RUN( *program* ) [, WAIT ]**

|                |                                                                                                                                |
|----------------|--------------------------------------------------------------------------------------------------------------------------------|
| <b>#RUN</b>    | Launches the named <i>program</i> .                                                                                            |
| <i>program</i> | A string constant containing the name of the program to execute, including any command line parameters.                        |
| <b>WAIT</b>    | Specifies waiting for the <i>program</i> to complete its execution before proceeding to the next template language statement.. |

The **#RUN** statement launches the specified *program* by using the Clarion language RUN statement. Template execution continues concurrently with the execution of the *program*, unless the **WAIT** attribute is specified.

Example:

```
#RUN('MyExe.EXE')
```

```
#!Launch MyExe.EXE
```

### #SERVICE (TopSpeed internal use only)

---

**#SERVICE**

The **#SERVICE** statement is for TopSpeed's internal use only. It is used to invoke a Clarion language .DLL and passes in a pointer to a highly volatile internal symbol table which can and does change with each release. Therefore, it is not documented and is listed here simply because it is visible in the shipping templates.

#RUNDLL is provided to give you similar capability.

## #RUNDLL (execute DLL procedure)

```
#RUNDLL(library, procedure [, parameter]) [, | RETAIN |] [, WIN32]
| RELEASE |
```

|                  |                                                                                                                                                                                                                                                                                                                 |
|------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>#RUNDLL</b>   | Executes a procedure from a Windows-standard Dynamic Link Library (.DLL).                                                                                                                                                                                                                                       |
| <i>library</i>   | A string constant containing the name of the .DLL file, including the extension. This must match the “bitness” of the Clarion environment. For Clarion 4, the environment is 16-bit. When the Clarion environment becomes 32-bit (in some release following Clarion 4), then the DLL will need to be 32-bit.    |
| <i>procedure</i> | A string constant containing the name of the procedure in the DLL file to execute. This must match the “public” (exported) name of the procedure. For Clarion DLLs, this means either the “mangled” name that appears in the .EXP file, or the value of the NAME attribute placed on the procedure’s prototype. |
| <i>parameter</i> | A string constant or expression containing a single parameter to pass to the <i>procedure</i> .                                                                                                                                                                                                                 |
| <b>RETAIN</b>    | Specifies that the <i>library</i> stays in memory after the <i>procedure</i> has completed execution.                                                                                                                                                                                                           |
| <b>RELEASE</b>   | Specifies that the <i>library</i> is unloaded from memory after the <i>procedure</i> has completed execution.                                                                                                                                                                                                   |
| <b>WIN32</b>     | Specifies that the <i>library</i> is a 32-bit DLL.                                                                                                                                                                                                                                                              |

The **#RUNDLL** statement executes the specified *procedure* from the specified *library*. The *library* .DLL is dynamically loaded before the *procedure* executes and unloaded after, unless the **RETAIN** attribute is specified.

If the **RETAIN** attribute is specified, another **#RUNDLL** statement with the **RELEASE** attribute and naming the same *library* must follow so that the .DLL is unloaded from memory. Failure to unload the .DLL can result in memory leaks. The **RETAIN** and **RELEASE** attributes can be nested as long as the **#RUNDLL** statements match up.

If a *parameter* is named, then the *procedure* must accept a single \*CSTRING parameter (only). If no *parameter* is named, then the *procedure* must not accept any parameters. In either case, the *procedure* cannot return a value. If the *parameter* named is a single-valued user-defined symbol, then any changes to the passed parameter’s value in the called *procedure* are reflected in that *parameter* symbol.

Example:

```
!A .DLL 'Services.DLL' contains these two procedures:
```

```
SoundBeep PROCEDURE
```

```
CODE
```

```
BEEP
```

```
AskForName PROCEDURE(*CSTRING Name)
```

```
Window WINDOW('Enter Name'),AT(,,260,100),SYSTEM,GRAY,AUTO
```

```
ENTRY(@s20),AT(4,4),USE(Name)
```

```
BUTTON('OK'),AT(200,4),USE(?ok),STD(STD:Close)
```

```
END
```

```
CODE
```

```
OPEN(Window)
```

```
ACCEPT
```

```
END
```

```
! The Template code that calls the procedures in Services.DLL:
```

```
#DECLARE (%UserName)
```

```
#SET (%UserName, 'Unknown')
```

```
#RUNDLL ('Services.DLL', 'SoundBeep'),RETAIN ! Use RETAIN to improve performance
```

```
#RUNDLL ('Services.DLL', 'AskForName', %UserName)
```

```
#RUNDLL ('Services.DLL', 'SoundBeep'),RELEASE ! RELEASE matches the RETAIN
```



# 20 - MISCELLANEOUS TEMPLATE STATEMENTS

## Miscellaneous Statements

### #! (template code comments)

---

*#! comments*

**#!** Initiates Template Language comments.

*comments* Any text.

**#!** initiates Template Language comments. All text to the right of the **#!** to the end of the text line is ignored by the Template file processor. **#!** comments are not included in the generated source code.

Example:

```
#! These are Template comments which
#! will not end up in the generated code
```

### #< (aligned target language comments)

---

*#<comments*

**#<** Initiates an aligned target language comment.

*comments* Any text. This must start with the target language comment initiator.

**#<** initiates a target language comment which is included in the generated source code. The comment is generated at the column position specified by the **#COMMENT** statement. If the column position is occupied, the comment is appended one space to the right of the generated source code statement. Any standard target language syntax comments without a preceding **#<** are included in the generated code at whatever column position they occupy in the template.

Example:

```
#COMMENT(50)
#! This Template file comment will not be in the generated code
#<! This is a Clarion comment which appears in the generated code in column 50
! This Clarion comment appears in the generated code in column 2
#<// This is a C++ comment which appears in the generated code beginning in column 50
// This C++ comment appears in the generated code in column 2
```

See Also:

**#COMMENT**

## #ASSERT (evaluate assumption)

**#ASSERT( *condition* [, *message* ] )**

|                  |                                                                                                                                                  |
|------------------|--------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>#ASSERT</b>   | Evaluates assumptions to detect unexpected errors.                                                                                               |
| <i>condition</i> | An expression that should always evaluate as true.                                                                                               |
| <i>message</i>   | A string constant containing the text to display if the <i>condition</i> is not true. If omitted, the text of the <i>condition</i> is displayed. |

**#ASSERT** evaluates assumptions to detect unexpected errors. **#ASSERT** evaluates the *condition* expression and displays the *message* if the *condition* is not true. This allows you to test the veracity of any programming assumptions you are making at any point in your template code.

Example:

```
#ASSERT(%False) #!Displays '%False'
#ASSERT(%False, 'Message Text') #!Displays 'Message Text'
```

## #CLASS (define a formula class)

**#CLASS( *string*, *description* )**

|                    |                                                                                                                                      |
|--------------------|--------------------------------------------------------------------------------------------------------------------------------------|
| <b>#CLASS</b>      | Defines a formula class.                                                                                                             |
| <i>string</i>      | A string constant containing the formula class.                                                                                      |
| <i>description</i> | A string expression containing the description of the formula class to display in the list of those available in the Formula Editor. |

The **#CLASS** statement defines a formula class for use in the Formula Editor. The Formula Class allows the Template to determine the precise logical position at which the formula appears in the generated source code.

Example:

```
#PROCEDURE(SomeProc,'An Example Template'),WINDOW
#CLASS('START','At beginning of procedure')
#CLASS('LOOP','In process loop')
#CLASS('END','At end of procedure')
%Procedure PROCEDURE
%ScreenStructure
CODE
#INSERT(%GenerateFormulas,'START') #!Generate START class formulas
OPEN(%Screen)
ACCEPT
 #INSERT(%GenerateFormulas,'LOOP') #!Generate LOOP class formulas
END
#INSERT(%GenerateFormulas,'END') #!Generate END class formulas
```

## #COMMENT (specify comment column)

---

### #COMMENT( *column* )

**#COMMENT**      Sets the default column number for aligned comments.  
*column*            A numeric constant in the range 1 to 255.

The **#COMMENT** statement sets the default column number in which Clarion comments prefaced with the **#<** statement will be generated by the Application Generator.

Example:

```
#COMMENT(50) #!Set comment column
IF Action = 1 #<!If adding a record
 SomeVariable = InitVariable
END
```

See Also:            #<

## #DEBUG (toggle debug generation)

---

### #DEBUG( *value* )

**#DEBUG**            Toggles debug generation on and off.  
*value*                A symbol containing zero (0) or one(1). When zero, debug generation is turned off. When one, debug generation is turned on.

The **#DEBUG** statement toggles debug generation on and off, and overrides the Debug Generation setting in the Application Options dialog. When this is checked, **#DEBUG** controls the specific sections to output to the text file, so the file can be restricted to a manageable size.

Example:

```
#SET(%DebugSet,1)
#DEBUG(%DebugSet) #!Set debug generation on
IF Action = 1 #<!If adding a record
 SomeVariable = InitVariable
END
#SET(%DebugSet,0)
#DEBUG(%DebugSet) #!Set debug generation off
```

## #ERROR (display source generation error)

---

### #ERROR( *message* )

|                |                                                                                                                               |
|----------------|-------------------------------------------------------------------------------------------------------------------------------|
| <b>#ERROR</b>  | Displays a source generation error.                                                                                           |
| <i>message</i> | A string constant, user-defined symbol, or expression containing an error message to display in the Source Generation window. |

**#ERROR** displays a *message* in the Source Generation window. This could be information for the user. It may also alert the user that they made some error which will cause the procedure Template to generate invalid source code which could create compiler errors.

When a **#ERROR** statement is encountered at source code generation time, its message is displayed. The user may choose to abort the compile and link process, or continue on to the compiler.

Example:

```
#PROCEDURE(SampleProc,'This is a sample procedure')
#PROMPT('Access Key',KEY),%SampleAccessKey
 #IF(%SampleAccessKey = %NULL) #!IF the user did not enter a Key
 #SET(%ErrorSymbol,(%Procedure & ' Access Key blank')
 #ERROR(%ErrorSymbol)
 #ERROR('This error is Fatal -- DO NOT CONTINUE')
 #ABORT
#ENDIF
```

## #EXPORT (export symbol to text)

---

**#EXPORT**( [ *symbol* ] )

**#EXPORT** Creates a .TXA text file from a *symbol*.  
*symbol* The template symbol to export (%Module, %Procedure, or %Program). If omitted, the current application is exported.

**#EXPORT** outputs .TXA script text for the *symbol* to the current output file (see #CREATE or #OPEN). This .TXA file may then be used for importing to other Clarion applications.

Example:

```
#OPEN('MyExp.TXA')
#FOR(%Procedure)
 #EXPORT(%Procedure)
#ENDFOR
```

See Also:           #CREATE, #OPEN, #IMPORT

## #HELP (specify template help file)

---

**#HELP**( *helpfile* )

**#HELP** Specifies the Template's help file.  
*helpfile* A string constant containing the name of the template's help file.

The **#HELP** statement specifies a *helpfile* which is used by this template. Once specified, the *helpfile* is used to access the help topics specified by the help id's in all HLP attributes in the template.

Example:

```
#HELP('Template.HLP')
```

## #INCLUDE (include a template file)

---

**#INCLUDE**( *filename* )

**#INCLUDE** Adds a template file to the Template file chain.  
*filename* A string constant containing the name of the template file to include.

The **#INCLUDE** statement adds a template file to the Template file chain. The template file containing the **#INCLUDE** statement continues to be processed after the included file has been processed.

Example:

```
#TEMPLATE(Clarion,'Clarion Standard Shipping Templates')
#include('Clarion1.TPX') #!Include a template file
#include('Clarion2.TPX') #!Include another template file
```

## #IMPORT (import from text script)

---

**#IMPORT**( *source* ) [ , | **RENAME** | ]  
 | **REPLACE** |

**#IMPORT** Creates an .APP for Clarion for Windows from a .TXA script *source* file.

*source* The name of the .TXA script file from which to create the .APP file.

**RENAME** Overrides the **Procedure Name Clash** prompt dialog and renames all procedures.

**REPLACE** Overrides the **Procedure Name Clash** prompt dialog and replaces all procedures.

**#IMPORT** adds procedure definitions to a Clarion for Windows .APP file from a .TXA script *source* file. This is used for importing from other versions of Clarion application development products.

Example:

```
#UTILITY(SomeUtility,'Some Utility Template')
#PROMPT('File to import',@s64),%ImportFile
#IMPORT(%ImportFile)
```

## #MESSAGE (display source generation message)

---

**#MESSAGE**( *message*, *line* )

|                 |                                                                                                                                                                                                            |
|-----------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>#MESSAGE</b> | Displays a source generation message.                                                                                                                                                                      |
| <i>message</i>  | A string constant, or a user-defined symbol, containing a message to display in the Source Generation dialog.                                                                                              |
| <i>line</i>     | An integer constant or symbol containing the line number on which to display the <i>message</i> . If out of the range 1 through 3, the <i>message</i> is displayed in the title bar as the window caption. |

**#MESSAGE** displays a *message* in the Source Generation message dialog. The first **#MESSAGE** statement displays the message window. Subsequent **#MESSAGE** statements modify the display text.

Example:

```
#MESSAGE('Generating ' & %Application,0) #!Display Title bar text
#MESSAGE('Generating ' & %Procedure,2) #!Display Progress message on line 2
```

## #PROTOTYPE (procedure prototype)

### #PROTOTYPE( *parameter list* )

**#PROTOTYPE** Assigns the *parameter list* to the **Prototype** entry field.  
*parameter list* A string constant containing the procedure's prototype parameter list (the entire procedure prototype without the leading procedure name) for the application's MAP structure (see the discussion of *Procedure Prototypes* in the *Language Reference*).

The **#PROTOTYPE** statement assigns the *parameter list* to the **Prototype** entry field on the Application Generator's Procedure Properties window, which automatically "dims out" the field (the programmer may not override this Prototype). This allows you to create procedure Templates which require a specific parameter list without forcing the programmer to know the procedure's prototype.

The **#PROTOTYPE** statement is valid only within a **#PROCEDURE** section and only one is allowed per **#PROCEDURE** section. If there is no **#PROTOTYPE** statement in the **#PROCEDURE**, the programmer is allowed to change it.

Example:

```
#PROCEDURE(SomeProc,'Some Procedure Template')
%Procedure PROCEDURE(Parm1,Parm2,Parm3)
 #PROTOTYPE('(STRING,*LONG,<*SHORT>')
 #!This procedure expects three parameters:
 #! a STRING passed by value
 #! a LONG passed by address
 #! a SHORT passed by address which may be omitted

#PROCEDURE(SomeFunc,'Some Template Procedure')
%Procedure PROCEDURE(Parm1,Parm2,Parm3)
 #PROTOTYPE('(STRING,*LONG,<*SHORT>),STRING')
 #!This procedure expects three parameters:
 #! a STRING passed by value
 #! a LONG passed by address
 #! a SHORT passed by address which may be omitted
 #!It returns a STRING
```

## #PROJECT (add file to project)

---

### #PROJECT( *module* )

|                 |                                                                                                                                                                                                                                                                                                                                            |
|-----------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>#PROJECT</b> | Includes a source or object code library, or Project file, in the application's Project file.                                                                                                                                                                                                                                              |
| <i>module</i>   | A string constant which names a source (.CLW, if Clarion is the target language), object (.OBJ), or library (.LIB) file containing procedures required by the procedure Template. This may also name a Project (.PRJ) file to be called by the application's Project. The type of file being imported is determined by the file extension. |

The **#PROJECT** statement specifies a source or object code library, or Project file, which is required to be in the application for the correct functioning of procedures created by the procedure Template.

**#PROJECT** provides a direct method of communicating *module* information to the Application Generator and Project system. It alerts the Application Generator to the required presence of the *module* for compiling and/or linking the application. Therefore, the application's Project file (generated by the Application Generator) automatically includes the *module* for making, compiling, and/or linking.

If multiple instances of the same **#PROJECT** statement are referenced by procedures created in the application, only the first is used. This would occur when multiple procedure Templates require the same *module*, or multiple application procedures are created from the same procedure Template.

**#PROJECT** allows a developer to automate the installation of third-party libraries and Templates to other developer's computers. This ensures that the application's Project is generated correctly.

**#PROJECT** for a Project (.PRJ) file provides the ability to create a hierarchy of Projects on large development projects. Where multiple libraries are being linked into a package, this allows you to ensure "make dependencies" are met for all libraries referenced in a particular project.

Example:

```
#AT(%CustomGlobalDeclarations)
 #PROJECT('Party3.LIB')
#ENDAT
```

## Built-in Template Procedures

### CALL (call a #GROUP as a function)

**CALL**( *symbol* [, *parameters* ] )

|                   |                                                                                                                                                                  |
|-------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>CALL</b>       | Calls a #GROUP as a function.                                                                                                                                    |
| <i>symbol</i>     | A symbol that names a #GROUP section in the current template set.                                                                                                |
| <i>parameters</i> | The parameters passed to the #GROUP. Each parameter must be separated by a comma. All parameters defined for the #GROUP must be passed; they may not be omitted. |

The **CALL** procedure places the return value from the #GROUP named by the *symbol* into the expression containing the **CALL**.

The *parameters* passed to the #GROUP fall into two categories: value-parameters and variable-parameters. Value-parameters are declared by the #GROUP as a user-defined symbol, while variable-parameters are declared by the #GROUP as a user-defined symbol with a prepended asterisk (\*). Either a symbol or an expression may be passed as a value-parameter. Only a symbol may be passed as a variable-parameter.

Return Data Type:      **STRING**

Example:

```
#SET(%SomeGroup, CALL(%SelectAGroup)) #!Call %SelectAGroup to get a #GROUP name
#INVOKE(%SomeGroup) #!Insert either %TrueGroup or %FalseGroup

#GROUP(%SelectAGroup)
#IF (%SomeCondition)
 #RETURN('%TrueGroup(Clarion)')
#ELSE
 #RETURN('%FalseGroup(Clarion)')
#ENDIF
```

See Also:              **#GROUP, INVOKE, #RETURN, #CALL, #INVOKE**

## EXTRACT (return attribute)

---

**EXTRACT**( *string*, *attribute* [, *parameter* ] )

|                  |                                                                                                                                                                                                                                                     |
|------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>EXTRACT</b>   | Returns the complete form of the specified <i>attribute</i> from the property <i>string</i> symbol.                                                                                                                                                 |
| <i>string</i>    | The symbol containing the properties to parse.                                                                                                                                                                                                      |
| <i>attribute</i> | A string constant or symbol containing the name of the property to return.                                                                                                                                                                          |
| <i>parameter</i> | An integer constant or symbol containing the number of the property's parameter to return. Zero (0) returns the entire parameter list (without the <i>attribute</i> ). If omitted, the <i>attribute</i> and all its <i>parameters</i> are returned. |

The **EXTRACT** procedure returns either the complete form of the specified *attribute* from the attribute *string* symbol, or just the specified *parameter*. This is useful if no built-in symbol exists for the particular attribute you need.

Return Data Type:      **STRING**

Example:

```
#SET(%MySymbol,EXTRACT(%ControlStatement,'DROPID') #!Return DROPID attribute
#SET(%MySymbol,EXTRACT(%ControlStatement,'DROPID',0) #!Return all DROPID parameters
```

See Also:              **REPLACE**

## EXISTS (return embed point existence)

---

### EXISTS( *symbol* )

#### **EXISTS**

Returns TRUE if the embedded source code point is available for use.

#### *symbol*

The *identifier* symbol for a #EMBED embedded source code point.

The **EXISTS** procedure returns true ('1') if the embedded source code point is available for use, at design-time only. If the embedded source code point is not available for use, EXISTS returns false (''). An embedded source code point is available for use if the section containing it is being used. This means that all #EMBEDs in the #PROCEDURE section, and all #GROUP sections referenced in the #PROCEDURE, are always available. #EMBEDs in a #CONTROL, #CODE, or #EXTENSION section are available only if the section is being used.

Return Data Type:      LONG

Example:

```
#IF(EXISTS(%CodeTemplateEmbed) = %True)
 !Generate some source
#ENDIF
```

## FILEEXISTS (return file existence)

---

### FILEEXISTS( *file* )

#### **FILEEXISTS**

Returns TRUE if the *file* is available on disk.

#### *file*

An expression containing the DOS filename.

The **FILEEXISTS** procedure returns true ('1') if the *file* is available on disk. If the *file* is not available, FILEEXISTS returns false ('').

Return Data Type:      LONG

Example:

```
#IF(FILEEXISTS(%SomeFile))
 #OPEN(%SomeFile)
 #READ(%SomeFile)
 !some source
#ENDIF
```

## FULLNAME (return file path)

---

**FULLNAME**( *file* )

**FULLNAME** Returns the fully qualified path of the specified *file* on disk.

*file* A string constant or expression containing the filename (including extension).

The **FULLNAME** procedure returns the fully qualified path of the specified *file* on disk.

Return Data Type:     **STRING**

Example:

```
#RUNDLL('Process File','MyModule.DLL',FULLNAME(%ChosenFile))
```

## INLIST (return item exists in list)

---

**INLIST**( *item*, *symbol* )

**INLIST** Returns the instance number of the *item* in the *symbol*.

*item* A string constant or symbol containing the name of the item to return.

*symbol* A multi-valued symbol that may contain the item.

The **INLIST** procedure returns the instance number of the *item* in the *symbol*. If the *item* is not contained in the *symbol*, **INLIST** returns zero (0).

Return Data Type:     **LONG**

Example:

```
#IF(INLIST('?MyControl',%Control))
 !Generate some source
#endif
```

## INSTANCE (return current instance number)

---

### INSTANCE( *symbol* )

**INSTANCE** Returns the current instance number to which the *symbol* is fixed.

*symbol* A multi-valued symbol.

The **INSTANCE** procedure returns the current instance number to which the *symbol* is fixed. If no **#FIX** or **#FOR** has been issued for the *symbol*, **INSTANCE** returns zero (0).

Return Data Type: **LONG**

Example:

```
#DELETE(%Control,INSTANCE(%Control)) #!Delete current instance
```

## INVOKE (call a named #GROUP as a function)

**INVOKE**( *symbol* [, *parameters* ] )

|                   |                                                                                                                                                                  |
|-------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>INVOKE</b>     | Calls a named #GROUP as a function.                                                                                                                              |
| <i>symbol</i>     | A symbol that contains the name of a #GROUP section in the current template set.                                                                                 |
| <i>parameters</i> | The parameters passed to the #GROUP. Each parameter must be separated by a comma. All parameters defined for the #GROUP must be passed; they may not be omitted. |

The **INVOKE** procedure places the return value from the #GROUP named in the *symbol* into the expression containing the INVOKE.

The *parameters* passed to the #GROUP fall into two categories: value-parameters and variable-parameters. Value-parameters are declared by the #GROUP as a user-defined symbol, while variable-parameters are declared by the #GROUP as a user-defined symbol with a prepended asterisk (\*). Either a symbol or an expression may be passed as a value-parameter. Only a symbol may be passed as a variable-parameter.

Return Data Type:      **STRING**

Example:

```
#SET(%SomeGroup, CALL(%SelectAGroup)) #!Call %SelectAGroup to get a #GROUP name
#SET(%AnotherGroup, INVOKE(%SomeGroup)) #!Call Selected Group for another #GROUP name
#INVOKE(%AnotherGroup) #!Insert either %DoTrueGroup or %DoFalseGroup

#GROUP(%SelectAGroup)
#IF (%SomeCondition)
 #RETURN('%TrueGroup(Clarion)')
#ELSE
 #RETURN('%FalseGroup(Clarion)')
#ENDIF

#GROUP(%TrueGroup)
 #RETURN('%DoTrueGroup(Clarion)')

#GROUP(%FalseGroup)
 #RETURN('%DoFalseGroup(Clarion)')
```

See Also:              **#GROUP, CALL, #RETURN, #CALL, #INVOKE**

## ITEMS (return multi-valued symbol instances)

---

**ITEMS**( *symbol* )

**ITEMS** Returns the number of instances contained by the *symbol*.

*symbol* A multi-valued symbol.

The **ITEMS** procedure returns the number of instances contained by the *symbol*.

Return Data Type: LONG

Example:

```
#DELETE(%Control,ITEMS(%Control)) #!Delete last instance
```

## LINKNAME (return mangled procedure name)

---

**LINKNAME**( *prototype* )

**LINKNAME** Returns the mangled procedure name for the linker.

*prototype* A string constant or symbol containing the prototype of the procedure as it appears in the MAP.

The **LINKNAME** procedure returns the mangled procedure name for the linker.

Return Data Type: STRING

Example:

```
%(LINKNAME(%Procedure & %Prototype)) @%ExpLineNumber
```

## QUOTE (replace string special characters)

**QUOTE**( *symbol* )

**QUOTE** Expands the *symbol*'s string data, “doubling up” single quotes (‘), and all un-paired left angle brackets (<) and left curly braces ({} to prevent compiler errors.

*symbol* The symbol containing the properties to parse.

The **QUOTE** procedure returns the string contained in the symbol with all single quotes (‘), un-paired left angle brackets (<), and un-paired left curly braces ({} “doubled up” to prevent compiler errors. This allows the user to enter string constants containing apostrophes, and filter expressions containing less than signs (<) without requiring that they enter two of each.

Return Data Type: **STRING**

Example:

```
#PROMPT('Filter Expression',@S255),%FilterExpression
#SET(%ValueConstruct,QUOTE(%FilterExpression)) #!Expand single quotes and angle
brackets
```

See Also: %', UNQUOTE

## REGISTERED (return template registration)

**REGISTERED**( *template* )

**REGISTERED** Returns TRUE if the *template* is available for use.

*template* A string constant containing the name of the #PROCEDURE, #CONTROL, #EXTENSION, #CODE, or #GROUP whose availability is in question.

The **REGISTERED** procedure returns true (‘1’) if the *template* is available for use, at design-time only. If the *template* is not available for use, REGISTERED returns false (‘’).

Return Data Type: **LONG**

Example:

```
#IF(NOT REGISTERED('BrowseBox(Clarion)'))
#ERROR('BrowseBox not registered')
#ENDIF
```

## REPLACE (replace attribute)

**REPLACE**( *string*, *attribute*, *new value* [, *parameter* ] )

|                  |                                                                                                                                                                                                                                                     |
|------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>REPLACE</b>   | Finds the complete form of the specified <i>attribute</i> from the property <i>string</i> symbol and replaces it with the <i>new value</i> .                                                                                                        |
| <i>string</i>    | The symbol containing the properties to parse.                                                                                                                                                                                                      |
| <i>attribute</i> | A string constant or symbol containing the name of the property to find.                                                                                                                                                                            |
| <i>new value</i> | A string constant or symbol containing the replacement value for the <i>attribute</i> .                                                                                                                                                             |
| <i>parameter</i> | An integer constant or symbol containing the number of the property's parameter to affect. Zero (0) affects the entire parameter list (without the <i>attribute</i> ). If omitted, the <i>attribute</i> and all its <i>parameters</i> are affected. |

The **REPLACE** procedure replaces either the complete form of the specified *attribute* from the attribute *string* symbol, or just the specified *parameter* with the *new value*. It returns the modified *string*.

Return Data Type: **STRING**

Example:

```
#SET(%ValueConstruct,REPLACE(%ValueConstruct,'MSG','')) #!Remove MSG attribute
```

See Also: **EXTRACT**

## SEPARATOR (return attribute string delimiter position)

**SEPARATOR**( *string*, *start* )

|                  |                                                                                                   |
|------------------|---------------------------------------------------------------------------------------------------|
| <b>SEPARATOR</b> | Returns the position of the next comma in the attribute <i>string</i> .                           |
| <i>string</i>    | A string constant or symbol containing a comma delimited list of attributes.                      |
| <i>start</i>     | An integer constant or symbol containing the starting position from which to seek the next comma. |

The **SEPARATOR** procedure returns the position of the next comma in the attribute *string* from the *start* position. This procedure correctly processes nested quotes within the *string* so that commas in string constants do not cause it to return an incorrect position.

Return Data Type:      LONG

Example:

```
#SET(%MySymbol,SEPARATOR(%ControlStatement,1))
 #!Return first comma position
```

## SLICE (return substring from string)

**SLICE**( *expression*, *start*, *end* )

|                   |                                                                                                                                              |
|-------------------|----------------------------------------------------------------------------------------------------------------------------------------------|
| <b>SLICE</b>      | Returns a substring portion of the <i>expression</i> string.                                                                                 |
| <i>expression</i> | A string constant, symbol, or expression containing the string from which to return a substring portion.                                     |
| <i>start</i>      | An integer constant or symbol containing the starting position (inclusive) within the <i>expression</i> from which to extract the substring. |
| <i>end</i>        | An integer constant or symbol containing the ending position (inclusive) within the <i>expression</i> from which to extract the substring.   |

The **SLICE** procedure returns the substring portion of the *expression* string identified by the *start* and *end* position values. This is equivalent to the Clarion language string slicing operation.

Return Data Type:      STRING

Example:

```
#SET(%MySymbol,SLICE('ABCDE',2,4))
 #!Return 'BCD'
```

## UNQUOTE (remove string special characters)

### UNQUOTE( *symbol* )

**UNQUOTE** Contracts the *symbol's* string data, “singling up” doubled single quotes (“), and all doubled un-paired left angle brackets (<<) and left curly braces ({}).

*symbol* The symbol containing the properties to parse.

The **UNQUOTE** procedure returns the string contained in the symbol with all doubled single quotes (“), doubled un-paired left angle brackets (<<), and un-paired left curly braces ({} “singled up” (returned to single instances instead of double instances of each character). This is the direct opposite to the **QUOTE** procedure.

Return Data Type: **STRING**

Example:

```
#PROMPT('Filter Expression',@S255),%FilterExpression
#SET(%ValueConstruct,QUOTE(%FilterExpression)) #!Expand single quotes and angle
brackets
#SET(%SingledValue,UNQUOTE(%ValueConstruct)) #!Contract single quotes and angle
brackets
```

See Also: %', QUOTE

## VAREXISTS (return symbol existence)

### VAREXISTS( *symbol* )

**EXISTS** Returns TRUE if the *symbol* is available for use.

*symbol* The symbol whose existence is in question.

The **VAREXISTS** procedure returns true ('1') if the *symbol* is available for use, at design-time only. If the *symbol* is not available for use, VAREXISTS returns false ('').

Return Data Type: **LONG**

Example:

```
#IF(NOT VAREXISTS(%MySymbol))
#DECLARE(%MySymbol)
#ENDIF
```

See Also: #DECLARE

# 21 - TEMPLATE SYMBOLS

## Symbol Overview

Clarion for Windows' Template Language uses symbols which act as variables do in a programming language -- they contain information that can be used as-is or may be used in expressions. These symbols may come from the built-in symbol set, or can be defined by the template author. Both types may be single-valued or multi-valued.

The built-in symbols that are available to the Template writer contain information from both the Dictionary and the Application about how the programmer has designed the application. The template-defined symbols contain information provided by the programmer from prompts on the Application Generator's properties windows, or may only be for internal use.

All template symbols expand during source generation to place the value they contain in the generated source code (if included in template code that generates source).

## Expansion Symbols

---

There are several special symbol forms that expand to allow formatting and special characters to generate into the source. These may be combined with each other to produce complex effects.

- |                                    |                                                                                                                                                                                  |
|------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| %%                                 | Expands to a single percent (%) sign. This allows the Application Generator to generate the modulus operator without confusion with any symbol.                                  |
| %#                                 | Expands to a single pound (#) sign. This allows the Application generator to generate an implicit LONG variable without confusion with any Template Language statement.          |
| % @ <i>picture</i> @ <i>symbol</i> | Formats the <i>symbol</i> with the specified <i>picture</i> when source generates. For example, % @D1 @MyDate expands the %MyDate symbol, formatted for the @D1 <i>picture</i> . |
| % [ <i>number</i> ] <i>symbol</i>  | Expands the <i>symbol</i> to fill at least the <i>number</i> of spaces specified. This allows proper comment and data type alignment in the generated source.                    |
| %                                  | Expands the next generated source onto the same line as the last. This is the Template line continuation character.                                                              |

|                            |                                                                                                                                                                                                                           |
|----------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>%'symbol</code>      | Expands the <i>symbol's</i> string data, “doubling up” single quotes ( <code>'</code> ), and all un-paired left angle brackets ( <code>&lt;</code> ) and left curly braces ( <code>{</code> ) to prevent compiler errors. |
| <code>%(expression)</code> | Expands the <i>expression</i> into the generated source.                                                                                                                                                                  |

Example:

```
%(ALL(' ', %indent))[20]Field %@d3@Date
 #!Generate an indent, expand %Field to occupy at least 20 spaces, then
 #! generate the date in mmm dd, yyyy format

%[30]Null #!Generate 30 spaces

#! %MySymbol contains: Gavin's Holiday
StringVar = '%MySymbol' #!Expands as a valid Clarion string constant
 #! to 'Gavin's Holiday'
```

## Symbol Hierarchy Overview

---

The Built-in Symbols all form a hierarchy of dependencies. This hierarchy starts with %Application, upon which all the other built-in symbols are dependent. The following tree diagram does not show all the dependent symbols, but does graphically represent the hierarchy of symbols. Most of these are multi-valued symbols.

- %Application**
  - %DictionaryFile**
    - %File**
      - %Field**
      - %Key**
      - %Relation**
  - %Program**
  - %GlobalData**
  - %Module**
    - %ModuleProcedure**
    - %MapItem**
    - %ModuleData**
  - %Procedure**
    - %Report**
      - %ReportControl**
        - %ReportControlField**
  - %Window**
    - %WindowEvent**
    - %Control**
      - %ControlEvent**
  - %ProcedureCalled**
  - %LocalData**
  - %ActiveTemplate**
    - %ActiveTemplateInstance**
  - %Formula**
    - %FormulaExpression**

These symbols (and all the symbols not listed here that are dependent upon these) contain all the information about the application that is available in the data dictionary (.DCT) and application (.APP) files. They enable you to write a template to generate any type of code you require.

## Built-in Symbols

### Symbols Dependent on %Application

---

|                                    |                                                                                                                                                                                            |
|------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| %Application                       | The name of the .APP file. The hierarchy of built-in symbols starts with %Application.                                                                                                     |
| %ApplicationDebug                  | Contains 1 if the application has debug enabled.                                                                                                                                           |
| %ApplicationLocalLibrary           | Contains 1 if the application is linking in the Clarion runtime library.                                                                                                                   |
| %ApplicationTemplate               | The name of all global extension templates used in the application. Multi-valued.                                                                                                          |
| %ApplicationTemplateInstance       | The instance numbers of all global extension templates used in the application. Multi-valued. Dependent on %ApplicationTemplate.                                                           |
| %ApplicationTemplateParentInstance | The instance number of the control template's parent global extension template. This is the global extension template that it is "attached" to. Dependent on %ApplicationTemplateInstance. |
| %ProjectTarget                     | Contains the name of the file being produced.                                                                                                                                              |
| %Target32                          | Contains 1 if the application is producing a 32-bit program.                                                                                                                               |
| %DictionaryChanged                 | Contains 1 if the .DCT file has changed since the last source generation.                                                                                                                  |
| %RegistryChanged                   | Contains 1 if the .REGISTRY.TRF file has changed since the last source generation.                                                                                                         |
| %ProgramDateCreated                | The program creation date (a Clarion standard date).                                                                                                                                       |
| %ProgramDateChanged                | The date the program was last changed (a Clarion standard date).                                                                                                                           |
| %ProgramTimeCreated                | The program creation time (Clarion standard time).                                                                                                                                         |
| %ProgramTimeChanged                | The time the program was last changed (a Clarion standard time).                                                                                                                           |

|                        |                                                                                                                                 |
|------------------------|---------------------------------------------------------------------------------------------------------------------------------|
| %FirstProcedure        | The label of the application's first procedure.                                                                                 |
| %HelpFile              | The name of the application's help file.                                                                                        |
| %ProgramExtension      | Contains EXE, DLL, or LIB.                                                                                                      |
| %DictionaryFile        | The name of the .DCT file for the application.                                                                                  |
| %DictionaryToolOptions | A string containing the entries third-party tools have made in the TOOLOPTIONS section of the .TXD file for the dictionary.     |
| %File                  | Contains all file declarations in the .DCT file. Multi-valued. Dependent on %DictionaryFile.                                    |
| %Program               | The name of the PROGRAM module.                                                                                                 |
| %GlobalData            | The labels of all global variable declarations made through the Global Data button on the Global Settings window. Multi-valued. |
| %GlobalDataStatement   | The variable's declaration statement (data type and all attributes). Dependent on %GlobalData.                                  |
| %GlobalDataLevel       | The indent level for formatting complex data structures. Dependent on %GlobalData.                                              |
| %Module                | The names of all source code modules. Multi-valued.                                                                             |
| %CreateLocalMap        | Contains 1 if the application should create local MAP structures instead of a single global MAP.                                |
| %QuickProcedure        | The name of the procedure type a #UTILITY with the WIZARD attribute is creating.                                                |
| %EditProcedure         | The name of the procedure being edited in context.                                                                              |
| %EditFilename          | The name of the temporary file that the procedure being edited in context is generated into.                                    |
| %Procedure             | The names of all procedures in the application. Multi-valued.                                                                   |

## **Symbols Dependent on %File**

---

|                  |                                                                                                       |
|------------------|-------------------------------------------------------------------------------------------------------|
| %File            | Contains all file declarations in the .DCT file. Multi-valued. Dependent on %DictionaryFile.          |
| %FileIdent       | The internal file number assigned by the Dictionary Editor and displayed in the IDENT in a .TXD file. |
| %FilePrefix      | Contents of the PRE attribute (the file prefix).                                                      |
| %FileDescription | A short description of the file.                                                                      |

|                      |                                                                                                                       |
|----------------------|-----------------------------------------------------------------------------------------------------------------------|
| %FileType            | Contains FILE, VIEW, or ALIAS.                                                                                        |
| %FileUsage           | Contains FILE, GLOBAL, or POOL.                                                                                       |
| %GlobalDataLast      | Contains 1 if the “Generate Last” option is set for %FileUsage = GLOBAL. Dependent on %File.                          |
| %FileDriver          | Contents of the DRIVER attribute first parameter.                                                                     |
| %FileDriverParameter | Contents of the DRIVER attribute second parameter.                                                                    |
| %FileName            | Contents of the FILE statement's NAME attribute.                                                                      |
| %FileOwner           | Contents of the OWNER attribute.                                                                                      |
| %FileCreate          | Contains 1 if the file has the CREATE attribute.                                                                      |
| %FileReclaim         | Contains 1 if the file has the RECLAIM attribute.                                                                     |
| %FileEncrypt         | Contains 1 if the file has the ENCRYPT attribute.                                                                     |
| %FileBindable        | Contains 1 if the file has the BINDABLE attribute.                                                                    |
| %FileLongDesc        | A long description of the file.                                                                                       |
| %File32BitOnly       | Contains 1 if the file can only be used in 32-bit applications.                                                       |
| %FileStruct          | The FILE statement (the label and all attributes).                                                                    |
| %FileStructEnd       | The keyword END.                                                                                                      |
| %FileStructRec       | The RECORD statement (including label and any attributes).                                                            |
| %FileStructRecEnd    | The keyword END.                                                                                                      |
| %FileStatement       | Contains the FILE statement's attributes (only).                                                                      |
| %FileThreaded        | Contains 1 if the file has the THREAD attribute.                                                                      |
| %FileExternal        | Contains 1 if the file has the EXTERNAL attribute.                                                                    |
| %FileExternalModule  | Contents of the file's EXTERNAL attribute parameter.                                                                  |
| %FilePrimaryKey      | The label of the file's primary key.                                                                                  |
| %FileQuickOptions    | A comma-delimited string containing the choices the user made on the Options tab for the file.                        |
| %FileUserOptions     | A string containing the entries the user made in the User Options text box on the Options tab for the file.           |
| %FileToolOptions     | A string containing the entries third-party tools have made in the TOOLOPTIONS section of the .TXD file for the file. |
| %ViewFilter          | Contents of the FILTER attribute.                                                                                     |
| %ViewStruct          | The VIEW statement (including the label and all attributes).                                                          |

|                                 |                                                                                                                 |
|---------------------------------|-----------------------------------------------------------------------------------------------------------------|
| <code>%ViewStructEnd</code>     | The keyword END.                                                                                                |
| <code>%ViewStatement</code>     | The VIEW statement's attributes (only).                                                                         |
| <code>%ViewPrimary</code>       | The label of the VIEW's primary file.                                                                           |
| <code>%ViewPrimaryFields</code> | The labels of all fields in the VIEW from the primary file. Multi-valued.                                       |
| <code>%ViewPrimaryField</code>  | Dependent on <code>%ViewPrimaryFields</code> . Contains the label of a field in the VIEW from the primary file. |
| <code>%ViewFiles</code>         | The labels of all files in the VIEW. Multi-valued.                                                              |
| <code>%AliasFile</code>         | The label of the ALIASed file.                                                                                  |
| <code>%Field</code>             | The labels of all fields in the file (including MEMO fields). Multi-valued.                                     |
| <code>%Key</code>               | The labels of all keys and indexes for the file. Multi-valued.                                                  |
| <code>%KeyOrder</code>          | The labels of all keys, indexes, and orders for the file. Multi-valued.                                         |
| <code>%Relation</code>          | The labels of all files that are related to the file. Multi-Valued.                                             |

## Symbols Dependent on `%ViewFiles`

---

|                                 |                                                                                                           |
|---------------------------------|-----------------------------------------------------------------------------------------------------------|
| <code>%ViewFiles</code>         | The labels of all files in the VIEW. Multi-valued. Dependent on <code>%File</code> .                      |
| <code>%ViewFileStruct</code>    | The JOIN statement for a secondary file in the VIEW.                                                      |
| <code>%ViewFileStructEnd</code> | The keyword END.                                                                                          |
| <code>%ViewFile</code>          | Contains the label of the file.                                                                           |
| <code>%ViewJoinedTo</code>      | The label of the file to which the file is JOINed.                                                        |
| <code>%ViewFileFields</code>    | The labels of all fields in the file used in the VIEW. Multi-valued.                                      |
| <code>%ViewFileField</code>     | Contains the label of the field in the file used in the VIEW. Dependent on <code>%ViewFileFields</code> . |

## Symbols Dependent on `%Field`

---

|                          |                                                                                                               |
|--------------------------|---------------------------------------------------------------------------------------------------------------|
| <code>%Field</code>      | The labels of all fields in the file (including MEMO fields). Multi-valued. Dependent on <code>%File</code> . |
| <code>%FieldIdent</code> | The internal field number assigned by the Dictionary Editor and displayed in the IDENT in a .TXD file.        |

|                      |                                                                            |
|----------------------|----------------------------------------------------------------------------|
| %FieldDerivedFrom    | Label of the field from which the field was derived (including prefix).    |
| %FieldDescription    | A short description of the field.                                          |
| %FieldLongDesc       | A long description of the field.                                           |
| %FieldFile           | The label of the file containing the field.                                |
| %FieldID             | Label of the field without prefix.                                         |
| %FieldDisplayPicture | Default display picture.                                                   |
| %FieldRecordPicture  | STRING field storage definition picture.                                   |
| %FieldDimension1     | Maximum value of first array dimension.                                    |
| %FieldDimension2     | Maximum value of second array dimension.                                   |
| %FieldDimension3     | Maximum value of third array dimension.                                    |
| %FieldDimension4     | Maximum value of fourth array dimension.                                   |
| %FieldHelpID         | Contents of the HLP attribute.                                             |
| %FieldName           | Contents of the field's NAME attribute.                                    |
| %FieldRangeLow       | The lower range of valid values for the field.                             |
| %FieldRangeHigh      | The upper range of valid values for the field.                             |
| %FieldType           | Data type of the field.                                                    |
| %FieldPlaces         | Number of decimal places for the field.                                    |
| %FieldMemoSize       | Maximum size of the MEMO.                                                  |
| %FieldMemoImage      | Contains 1 if the MEMO has a BINARY attribute.                             |
| %FieldInitial        | Initial value for the field.                                               |
| %FieldLookup         | File to access to validate this field's value.                             |
| %FieldStruct         | The field's declaration statement (label , data type, and all attributes). |
| %FieldStatement      | The field's declaration statement (data type and all attributes).          |
| %FieldHeader         | The field's default report column header.                                  |
| %FieldPicture        | Default display picture.                                                   |
| %FieldJustType       | Contains L, R, C, or D for the field's justification.                      |
| %FieldJustIndent     | The justification offset amount.                                           |
| %FieldFormatWidth    | The default width for the field's ENTRY control.                           |

|                                        |                                                                                                                                                                                  |
|----------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>%FieldScreenControl</code>       | The field's screen control as defined in the data dictionary. Multi-valued.                                                                                                      |
| <code>%FieldScreenControlWidth</code>  | Either the default from the runtime library, or the explicitly set width in the data dictionary for the field's screen control. Dependent on <code>%FieldScreenControl</code> .  |
| <code>%FieldScreenControlHeight</code> | Either the default from the runtime library, or the explicitly set height in the data dictionary for the field's screen control. Dependent on <code>%FieldScreenControl</code> . |
| <code>%FieldReportControl</code>       | The field's report control as defined in the data dictionary. Multi-valued.                                                                                                      |
| <code>%FieldReportControlWidth</code>  | Either the default from the runtime library, or the explicitly set width in the data dictionary for the field's report control. Dependent on <code>%FieldReportControl</code> .  |
| <code>%FieldReportControlHeight</code> | Either the default from the runtime library, or the explicitly set height in the data dictionary for the field's report control. Dependent on <code>%FieldReportControl</code> . |
| <code>%FieldValidation</code>          | The choice the user made on the Validity Checks tab for the field. Can contain either NONZERO, INRANGE, BOOLEAN, INFILE, or INLIST.                                              |
| <code>%FieldChoices</code>             | The choices the user entered for a Must Be In List field ( <code>%FieldValidation</code> contains INLIST). Multi-valued.                                                         |
| <code>%FieldValues</code>              | The choices the user entered to override the VALUE attribute for the <code>%FieldChoices</code> symbol. Multi-valued.                                                            |
| <code>%FieldTrueValue</code>           | The true choice the user entered to override the VALUE attribute ( <code>%FieldValidation</code> contains BOOLEAN).                                                              |
| <code>%FieldFalseValue</code>          | The false choice the user entered to override the VALUE attribute ( <code>%FieldValidation</code> contains BOOLEAN).                                                             |
| <code>%FieldDisplayChoices</code>      | The choices the user entered in the display override field for a Must Be In List field ( <code>%FieldValidation</code> contains INLIST). Multi-valued.                           |
| <code>%FieldQuickOptions</code>        | A comma-delimited string containing the choices the user made on the Options tab for the field.                                                                                  |
| <code>%FieldUserOptions</code>         | A string containing the entries the user made in the User Options text box of the Options tab for the field.                                                                     |

**%FieldToolOptions** A string containing the entries third-party tools have made in the TOOLOPTIONS section of the .TXD file for the field.

## Symbols Dependent on %Key and %KeyOrder

---

|                          |                                                                                                                      |
|--------------------------|----------------------------------------------------------------------------------------------------------------------|
| <b>%Key</b>              | The labels of all keys and indexes for the file. Multi-valued.                                                       |
| <b>%KeyId</b>            | The internal key number assigned by the Dictionary Editor and displayed in the IDENT in a .TXD file.                 |
| <b>%KeyDescription</b>   | A short description of the key.                                                                                      |
| <b>%KeyLongDesc</b>      | A long description of the key.                                                                                       |
| <b>%KeyFile</b>          | The label of the file to which the key belongs.                                                                      |
| <b>%KeyID</b>            | The label of the key (without prefix).                                                                               |
| <b>%KeyIndex</b>         | Contains KEY, INDEX, or DYNAMIC.                                                                                     |
| <b>%KeyName</b>          | Contents of the key's NAME attribute.                                                                                |
| <b>%KeyAuto</b>          | Contains the label of the auto-incrementing field.                                                                   |
| <b>%KeyDuplicate</b>     | Contains 1 if the key has the DUP attribute.                                                                         |
| <b>%KeyExcludeNulls</b>  | Contains 1 if the key has the OPT attribute.                                                                         |
| <b>%KeyNoCase</b>        | Contains 1 if the key has the NOCASE attribute.                                                                      |
| <b>%KeyPrimary</b>       | Contains 1 if the key is the file's primary key.                                                                     |
| <b>%KeyStruct</b>        | The key's declaration statement (label and all attributes).                                                          |
| <b>%KeyStatement</b>     | The key's attributes (only).                                                                                         |
| <b>%KeyField</b>         | The labels of all component fields of the key. Multi-valued.                                                         |
| <b>%KeyFieldSequence</b> | Contains ASCENDING or DESCENDING. Dependent on %Keyfield.                                                            |
| <b>%KeyQuickOptions</b>  | A comma-delimited string containing the choices the user made on the Options tab for the key.                        |
| <b>%KeyUserOptions</b>   | A string containing the entries the user made in the User Options text box on the Options tab for the key.           |
| <b>%KeyToolOptions</b>   | A string containing the entries third-party tools have made in the TOOLOPTIONS section of the .TXD file for the key. |
| <b>%KeyOrder</b>         | The labels of all keys, indexes, and orders for the file. Multi-valued.                                              |
| <b>%KeyOrderIdent</b>    | The internal key number assigned by the Dictionary Editor and displayed in the IDENT in a .TXD file.                 |

|                                    |                                                                                                                      |
|------------------------------------|----------------------------------------------------------------------------------------------------------------------|
| <code>%KeyOrderDescription</code>  | A short description of the key.                                                                                      |
| <code>%KeyOrderLongDesc</code>     | A long description of the key.                                                                                       |
| <code>%KeyOrderFile</code>         | The label of the file to which the key belongs.                                                                      |
| <code>%KeyOrderID</code>           | The label of the key (without prefix).                                                                               |
| <code>%KeyOrderIndex</code>        | Contains KEY, INDEX, or DYNAMIC.                                                                                     |
| <code>%KeyOrderName</code>         | Contents of the key's NAME attribute.                                                                                |
| <code>%KeyOrderAuto</code>         | Contains the label of the auto-incrementing field.                                                                   |
| <code>%KeyOrderDuplicate</code>    | Contains 1 if the key has the DUP attribute.                                                                         |
| <code>%KeyOrderExcludeNulls</code> | Contains 1 if the key has the OPT attribute.                                                                         |
| <code>%KeyOrderNoCase</code>       | Contains 1 if the key has the NOCASE attribute.                                                                      |
| <code>%KeyOrderPrimary</code>      | Contains 1 if the key is the file's primary key.                                                                     |
| <code>%KeyOrderStruct</code>       | The key's declaration statement (label and all attributes).                                                          |
| <code>%KeyOrderStatement</code>    | The key's attributes (only).                                                                                         |
| <code>%KeyOrderQuickOptions</code> | A comma-delimited string containing the choices the user made on the Options tab for the key.                        |
| <code>%KeyOrderUserOptions</code>  | A string containing the entries the user made in the User Options text box on the Options tab for the key.           |
| <code>%KeyOrderToolOptions</code>  | A string containing the entries third-party tools have made in the TOOLOPTIONS section of the .TXD file for the key. |

## Symbols Dependent on %Relation

---

|                                |                                                                                                 |
|--------------------------------|-------------------------------------------------------------------------------------------------|
| <code>%Relation</code>         | The labels of all files that are related to the file. Multi-Valued.                             |
| <code>%RelationPrefix</code>   | The prefix of the related file.                                                                 |
| <code>%RelationAlias</code>    | Contains the name of the file being aliased if the current file named in %Relation is an alias. |
| <code>%FileRelationType</code> | Contains 1:MANY or MANY:1.                                                                      |
| <code>%RelationKey</code>      | The label of the related file's linking key.                                                    |
| <code>%FileKey</code>          | The label of the file's linking key.                                                            |

|                                        |                                                                                                                           |
|----------------------------------------|---------------------------------------------------------------------------------------------------------------------------|
| <code>%RelationConstraintDelete</code> | May contain: RESTRICT, CASCADE, or CLEAR.                                                                                 |
| <code>%RelationConstraintUpdate</code> | May contain: RESTRICT, CASCADE, or CLEAR.                                                                                 |
| <code>%RelationKeyField</code>         | The labels of all linking fields in the related file's key. Multi-valued.                                                 |
| <code>%RelationKeyFieldLink</code>     | The label of the linking field in the file's key. Dependent on <code>%RelationKeyField</code> .                           |
| <code>%FileKeyField</code>             | The labels of all linking fields in the file's key. Multi-valued.                                                         |
| <code>%FileKeyFieldLink</code>         | The label of the linking field in the related file's key. Dependent on <code>%FileKeyField</code> .                       |
| <code>%RelationQuickOptions</code>     | A comma-delimited string containing the choices the user made on the Options tab for the relation.                        |
| <code>%RelationUserOptions</code>      | A string containing the entries the user made in the User Options text box on the Options tab for the relation.           |
| <code>%RelationToolOptions</code>      | A string containing the entries third-party tools have made in the TOOLOPTIONS section of the .TXD file for the relation. |

## Symbols Dependent on %Module

---

|                               |                                                                                                              |
|-------------------------------|--------------------------------------------------------------------------------------------------------------|
| <code>%Module</code>          | Thes names of all source code modules. Multi-valued.<br><code>%ModuleDescription</code> <code>%Module</code> |
| <code>%ModuleLanguage</code>  | Contains the module target language.                                                                         |
| <code>%ModuleTemplate</code>  | The name of the Module Template used to generate the module.                                                 |
| <code>%ModuleChanged</code>   | Contains 1 if anything in the module has changed since the last source generation.                           |
| <code>%ModuleExternal</code>  | Contains 1 if the module is external. (not generated by Clarion for Windows).                                |
| <code>%ModuleReadOnly</code>  | Contains 1 if the module is read only.                                                                       |
| <code>%ModuleExtension</code> | The file extension for the module.                                                                           |
| <code>%ModuleBase</code>      | The name of the module (without extension).                                                                  |
| <code>%ModuleInclude</code>   | The fiel to INCLUDE in the program MAP containing the module's prototypes.                                   |

|                                   |                                                                                                                            |
|-----------------------------------|----------------------------------------------------------------------------------------------------------------------------|
| <code>%ModuleProcedure</code>     | The names of all procedures in the module. Multi-valued.                                                                   |
| <code>%ModuleData</code>          | The labels of all module variable declarations made through the Data button on the Module Properties window. Multi-valued. |
| <code>%ModuleDataStatement</code> | The variable's declaration statement (data type and all attributes). Dependent on <code>%ModuleData</code> .               |

## Symbols Dependent on `%Procedure`

---

|                                    |                                                                                                           |
|------------------------------------|-----------------------------------------------------------------------------------------------------------|
| <code>%Procedure</code>            | The names of all procedures in the application. Multi-valued.                                             |
| <code>%ProcedureToDo</code>        | Contains 1 if the procedure is "ToDo." Dependent on <code>%Procedure</code> .                             |
| <code>%ProcedureType</code>        | Contains PROCEDURE or FUNCTION. Dependent on <code>%Procedure</code> .                                    |
| <code>%ProcedureReturnType</code>  | The data type returned, if the procedure is a FUNCTION. Dependent on <code>%Procedure</code> .            |
| <code>%ProcedureDateCreated</code> | The procedure creation date (a Clarion standard date). Dependent on <code>%Procedure</code> .             |
| <code>%ProcedureDateChanged</code> | The date the procedure was last changed (a Clarion standard date). Dependent on <code>%Procedure</code> . |
| <code>%ProcedureTimeCreated</code> | The time the procedure was created (a Clarion standard time). Dependent on <code>%Procedure</code> .      |
| <code>%ProcedureTimeChanged</code> | The time the procedure was last changed (a Clarion standard time). Dependent on <code>%Procedure</code> . |
| <code>%ProcedureReadOnly</code>    | Contains 1 if the procedure is read only. Dependent on <code>%Procedure</code> .                          |
| <code>%ProcedureIsGlobal</code>    | Contains 1 if the procedure is to be declared in the global MAP. Dependent on <code>%Procedure</code> .   |
| <code>%Prototype</code>            | The procedure's prototype for the MAP structure. Dependent on <code>%Procedure</code> .                   |

- %ProcedureTemplate**  
The name of the Procedure Template used to generate the procedure. Dependent on %Procedure.
- %ProcedureDescription**  
A short description of the procedure. Dependent on %Procedure.
- %ProcedureExported**  
Contains 1 if the procedure is in a DLL and is callable from outside the DLL. Dependent on %Procedure.
- %ProcedureLongDescription**  
A long description of the procedure. Dependent on %Procedure.
- %ProcedureLanguage**  
The target language the procedure template generates. Dependent on %Procedure.
- %ProcedureCalled** The names of all procedures listed by the Procedures button on the Procedure Properties window. Multi-valued. Dependent on %Procedure.
- %LocalData** The labels of all local variable declarations made through the Data button on the Procedure Properties window. Multi-valued. Dependent on %Procedure.
- %LocalDataStatement**  
The variable's declaration statement (data type and all attributes). Dependent on %LocalData.
- %LocalDataDescription**  
A short description of the field. Dependent on %LocalData.
- %LocalDataHeader** The field's default report column header. Dependent on %LocalData.
- %LocalDataPicture** Default display picture. Dependent on %LocalData.
- %LocalDataJustType**  
Contains L, R, C, or D for the field's justification. Dependent on %LocalData.
- %LocalDataJustIndent**  
The justification offset amount. Dependent on %LocalData.
- %LocalDataFormatWidth**  
The default width for the field's ENTRY control. Dependent on %LocalData.
- %ActiveTemplate** The names of all control templates used in the procedure. Multi-valued. Dependent on %Procedure.

- %ActiveTemplateInstance**  
The instance numbers of all control templates used in the procedure. Multi-valued. Dependent on %ActiveTemplate.
- %ActiveTemplateOwnerInstance**  
The instance number of the extension template which auto-populated the current extension template into a procedure through use of the parameter to the APPLICATION attribute on a #EXTENSION statement. Dependent on %ActiveTemplateInstance.
- %ActiveTemplateParentInstance**  
The instance number of the control template's parent control template. This is the control template that it is "attached" to. Dependent on %ActiveTemplateInstance.
- %ActiveTemplatePrimaryInstance**  
The instance number of the control template's primary control template. This is the first control template in a succession of multiple related control templates. Dependent on %ActiveTemplateInstance.

## Window Control Symbols

---

- %Window** The label of the procedure's window. Dependent on %Procedure.
- %WindowStatement**  
The WINDOW or APPLICATION declaration statement (and all attributes). Dependent on %Window.
- %MenuBarStatement**  
The MENUBAR declaration statement (and all attributes). Dependent on %Window.
- %ToolBarStatement** The TOOLBAR declaration statement (and all attributes). Dependent on %Window.
- %WindowEvent** All field-independent events, as listed in the EQUATES.CLW file (without EVENT: prepended). Multi-valued. Dependent on %Window.
- %Control** The field equate labels of all controls in the window. Multi-valued. Dependent on %Window.
- %ControlUse** The control's USE variable (not field equate). Dependent on %Control.
- %ControlStatement** The control's declaration statement (and all attributes). This may contain multiple lines of code if the declaration is too long to fit on a single line. Dependent on %Control.

|                                       |                                                                                                                                                         |
|---------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>%ControlUnsplitStatement</code> | The control's declaration statement (and all attributes) without line splitting. Dependent on <code>%Control</code> .                                   |
| <code>%ControlType</code>             | The type of control (MENU, ITEM, ENTRY, BUTTON, etc.). Dependent on <code>%Control</code> .                                                             |
| <code>%ControlTemplate</code>         | The name of the control template which populated the control onto the window. Dependent on <code>%Control</code> .                                      |
| <code>%ControlTool</code>             | Contains 1 if the control is in a TOOLBAR. Dependent on <code>%Control</code> .                                                                         |
| <code>%ControlParent</code>           | Contains the field equate label of the control's parent, if it is in a control structure (OPTION, GROUP, etc.). Dependent on <code>%Control</code> .    |
| <code>%ControlParentType</code>       | Contains the type of control of the control's parent, if it is in a control structure (OPTION, GROUP, etc.). Dependent on <code>%ControlParent</code> . |
| <code>%ControlParentTab</code>        | Contains the field equate label of the TAB on which the control has been placed. Dependent on <code>%Control</code> .                                   |
| <code>%ControlParameter</code>        | The control's parameter (the value ion the parentheses). Dependent on <code>%Control</code> .                                                           |
| <code>%ControlDefaultWidth</code>     | The control's estimated default width. Dependent on <code>%Control</code> .                                                                             |
| <code>%ControlDefaultHeight</code>    | The control's estimated default height. Dependent on <code>%Control</code> .                                                                            |
| <code>%ControlMenu</code>             | Contains 1 if the control is in a MENUBAR. Dependent on <code>%Control</code> .                                                                         |
| <code>%ControlToolBar</code>          | Contains the TOOLBAR declaration statement (and all attributes) if the control is the first control in a toolbar. Dependent on <code>%Control</code> .  |
| <code>%ControlMenuBar</code>          | Contains the MENUBAR declaration statement (and all attributes) if the control is the first control in a menu. Dependent on <code>%Control</code> .     |
| <code>%ControlIndent</code>           | The control declaration's indentation level in the generated data structure. Dependent on <code>%Control</code> .                                       |
| <code>%ControlInstance</code>         | The instance number of the control template which populated the control onto the window. Dependent on <code>%Control</code> .                           |
| <code>%ControlOriginal</code>         | The original field equate label of the control as listed in the control template from which it came. Dependent on <code>%Control</code> .               |

|                                      |                                                                                                                                                                             |
|--------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>%ControlFrom</code>            | The FROM attribute of a LIST or COMBO control. Dependent on <code>%Control</code> .                                                                                         |
| <code>%ControlAlert</code>           | All ALERT attributes for the control. Multi-valued. Dependent on <code>%Control</code> .                                                                                    |
| <code>%ControlEvent</code>           | All field-specific events appropriate for the control, as listed in the EQUATES.CLW file (without the EVENT: prepended). Multi-valued. Dependent on <code>%Control</code> . |
| <code>%ControlField</code>           | All fields populated into the LIST, COMBO, or SPIN control. Multi-valued. Dependent on <code>%Control</code> .                                                              |
| <code>%ControlFieldHasIcon</code>    | Contains 1 if the field in the LIST or COMBO control is formatted to have an icon. Dependent on <code>%ControlField</code> .                                                |
| <code>%ControlFieldHasColor</code>   | Contains 1 if the field in the LIST or COMBO control is formatted to have colors. Dependent on <code>%ControlField</code> .                                                 |
| <code>%ControlFieldHasTree</code>    | Contains 1 if the field in the LIST or COMBO control is formatted to be a tree. Dependent on <code>%ControlField</code> .                                                   |
| <code>%ControlFieldHasLocator</code> | Contains 1 if the field in the LIST or COMBO control is formatted to be a locator. Dependent on <code>%ControlField</code> .                                                |
| <code>%ControlFieldPicture</code>    | Contains the picture token of the field in the LIST or COMBO control. Dependent on <code>%ControlField</code> .                                                             |
| <code>%ControlFieldHeading</code>    | Contains the heading text of the field in the LIST or COMBO control. Dependent on <code>%ControlField</code> .                                                              |
| <code>%ControlFieldFormat</code>     | Contains the portion of the FORMAT attribute string that applies to the field in the LIST or COMBO control. Dependent on <code>%ControlField</code> .                       |

## Report Control Symbols

---

|                                |                                                                                                          |
|--------------------------------|----------------------------------------------------------------------------------------------------------|
| <code>%Report</code>           | The label of the procedure's report. Dependent on <code>%Procedure</code> .                              |
| <code>%ReportStatement</code>  | The REPORT declaration statement (and all attributes). Dependent on <code>%Report</code> .               |
| <code>%ReportControl</code>    | The field equate labels of all controls in the report. Multi-valued. Dependent on <code>%Report</code> . |
| <code>%ReportControlUse</code> | The control's USE variable (not field equate). Dependent on <code>%ReportControl</code> .                |

|                                      |                                                                                                                                                 |
|--------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>%ReportControlStatement</code> | The control's declaration statement (and all attributes).<br>Dependent on <code>%ReportControl</code> .                                         |
| <code>%ReportControlType</code>      | The type of control (MENU, ITEM, ENTRY, BUTTON, etc.). Dependent on <code>%ReportControl</code> .                                               |
| <code>%ReportControlTemplate</code>  | The name of the control template which populated the control onto the report. Dependent on <code>%ReportControl</code> .                        |
| <code>%ReportControlIndent</code>    | The control declaration's indentation level in the generated data structure. Dependent on <code>%ReportControl</code> .                         |
| <code>%ReportControlInstance</code>  | The instance number of the control template which populated the control onto the report. Dependent on <code>%ReportControl</code> .             |
| <code>%ReportControlOriginal</code>  | The original field equate label of the control as listed in the control template from which it came. Dependent on <code>%ReportControl</code> . |
| <code>%ReportControlLabel</code>     | The label of the report STRING control. Dependent on <code>%ReportControl</code> .                                                              |
| <code>%ReportControlField</code>     | All fields populateed into the LIST, COMBO, or SPIN control. Multi-valued. Dependent on <code>%ReportControl</code> .                           |

## Formula Symbols

---

|                                  |                                                                                                                                             |
|----------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------|
| <code>%Formula</code>            | The label of the result field for each formula. Multi-valued. Dependent on <code>%Procedure</code> .                                        |
| <code>%FormulaDescription</code> | A description of the formula.                                                                                                               |
| <code>%FormulaClass</code>       | An identifier for the position in generated source to place the formula.                                                                    |
| <code>%FormulaInstance</code>    | The control template instance number for a formula whose class has been declared in a control template.                                     |
| <code>%FormulaExpression</code>  | The expression to conditionally evaluate or assign to the result field for each formula. Multi-valued. Dependent on <code>%Formula</code> . |

|                                      |                                                                                                                           |
|--------------------------------------|---------------------------------------------------------------------------------------------------------------------------|
| <code>%FormulaExpressionType</code>  | Contains =, IF, ELSE, CASE, or OF. Dependent on <code>%FormulaExpression</code> .                                         |
| <code>%FormulaExpressionTrue</code>  | Contains the line number of the true expression in the generated formula. Dependent on <code>%FormulaExpression</code> .  |
| <code>%FormulaExpressionFalse</code> | Contains the line number of the false expression in the generated formula. Dependent on <code>%FormulaExpression</code> . |
| <code>%FormulaExpressionOf</code>    | Contains the line number of the OF expression in the generated formula. Dependent on <code>%FormulaExpression</code> .    |
| <code>%FormulaExpressionCase</code>  | Contains the line number of the assignment in the generated formula. Dependent on <code>%FormulaExpression</code> .       |

## File Schematic Symbols

---

|                                   |                                                                                                                                                                                             |
|-----------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>%Primary</code>             | The label of a Primary file listed in the procedure's File Schematic for the procedure or a control template used in the procedure.                                                         |
| <code>%PrimaryKey</code>          | The label of the access key for the primary file. Dependent on <code>%Primary</code> .                                                                                                      |
| <code>%PrimaryInstance</code>     | The control template instance number for which the file is primary. Dependent on <code>%Primary</code> .                                                                                    |
| <code>%Secondary</code>           | The labels of all Secondary files listed in the File Schematic for the procedure or a control template used in the procedure. Multi-valued. Dependent on <code>%Primary</code> .            |
| <code>%SecondaryTo</code>         | The label of the Secondary or Primary file to which the Secondary file is related (the file "above" it as listed in the procedure's File Schematic). Dependent on <code>%Secondary</code> . |
| <code>%SecondaryType</code>       | Contains 1:MANY or MANY:1. Dependent on <code>%Secondary</code> .                                                                                                                           |
| <code>%SecondaryCustomJoin</code> | Contains 1 if the JOIN is a custom join. Dependent on <code>%Secondary</code> .                                                                                                             |
| <code>%SecondaryCustomText</code> | Contains the text defining the custom join. Dependent on <code>%Secondary</code> .                                                                                                          |
| <code>%OtherFiles</code>          | The labels of all Other Data files listed for the procedure. Multi-valued.                                                                                                                  |

## File Driver Symbols

---

|                    |                                                                                              |
|--------------------|----------------------------------------------------------------------------------------------|
| %Driver            | The names of all registered file drivers.                                                    |
| %DriverDLL         | The name of the driver's .DLL file. Dependent on %Driver.                                    |
| %DriverLIB         | The name of the driver's .LIB file. Dependent on %Driver.                                    |
| %DriverDescription | A description of the file driver. Dependent on %Driver.                                      |
| %DriverCreate      | Contains 1 if the driver supports the CREATE attribute. Dependent on %Driver.                |
| %DriverOwner       | Contains 1 if the driver supports the OWNER attribute. Dependent on %Driver.                 |
| %DriverEncrypt     | Contains 1 if the driver supports the ENCRYPT attribute. Dependent on %Driver.               |
| %DriverReclaim     | Contains 1 if the driver supports the RECLAIM attribute. Dependent on %Driver.               |
| %DriverMaxKeys     | The maximum number of keys the driver supports for each data file. Dependent on %Driver.     |
| %DriverUniqueKey   | Contains 1 if the driver supports unique (no DUP attribute) keys. Dependent on %Driver.      |
| %DriverRequired    | Contains 1 if the driver supports the OPT attribute. Dependent on %Driver.                   |
| %DriverMemo        | Contains 1 if the driver supports MEMO fields. Dependent on %Driver.                         |
| %DriverBinMemo     | Contains 1 if the driver supports the BINARY attribute on MEMO fields. Dependent on %Driver. |
| %DriverSQL         | Contains 1 if the driver is an SQL driver. Dependent on %Driver.                             |
| %DriverType        | All data types supported by the driver. Multi-valued. Dependent on %Driver.                  |
| %DriverOpcode      | All operations supported by the driver. Multi-valued. Dependent on %Driver.                  |

## Miscellaneous Symbols

---

|                                   |                                                                                                                                                                  |
|-----------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>%ConditionalGenerate</code> | Contains 1 if the Conditional Generation box is checked on the Application Options window.                                                                       |
| <code>%CWVersion</code>           | Contains the current version release number of Clarion. For example, for the first release of version 2.0, this contains 2000.                                   |
| <code>%Null</code>                | Contains nothing. This is used for comparison to detect empty symbols.                                                                                           |
| <code>%True</code>                | Contains 1.                                                                                                                                                      |
| <code>%False</code>               | Contains an empty string ('').                                                                                                                                   |
| <code>%EOF</code>                 | Contains the value that flags the end of file when reading a file with <code>#READ</code> .                                                                      |
| <code>%BytesOutput</code>         | Contains the number of bytes written to the current output file. This can be used to detect empty embed points (if no bytes were written, it contained nothing). |
| <code>%EmbedID</code>             | Contains the current embed point's identifying symbol.                                                                                                           |
| <code>%EmbedDescription</code>    | The current embed point's description.                                                                                                                           |
| <code>%EmbedParameters</code>     | The current embed point's current instance, as a comma-delimited list.                                                                                           |
| <code>%Win32</code>               | Contains 1 if <code>#RUNDLL</code> with the WIN32 attribute can be executed.                                                                                     |



## 22 - ANNOTATED EXAMPLE TEMPLATES

### Procedure Template: Window

The *Window* Procedure template is the generic template that creates any window handling procedure. Since most (if not all) procedures in a Windows application have a window, the type of code this template generates forms the basis of the generated source code for most procedures.

The Window template is also the fundamental template upon which all the other Procedure templates are built. For example, the Browse template is actually a Window template with BrowseBox and BrowseUpdateButtons Control templates pre-defined for the procedure.

The following template language code is all the code for the Window Template:

**NOTE:** For this and all other code examples in this book, the template line continuation character (%) is used to split code lines that are too long to fit on the page. In the template files on disk these characters are not (and should not be) used to continue a template code line; they are used here only for readability. **All these examples are from the “Clarion” template chain.**

```
#PROCEDURE(Window,'Generic Window Handler'),WINDOW,HLP('~TPLProcWindow')
#LOCALDATA
LocalRequest LONG,AUTO
OriginalRequest LONG,AUTO
LocalResponse LONG,AUTO
WindowOpened LONG
WindowInitialized LONG
ForceRefresh LONG,AUTO
#ENDLOCALDATA
#CLASS('Procedure Setup','Upon Entry into the Procedure')
#CLASS('Before Lookups','Refresh Window ROUTINE, before lookups')
#CLASS('After Lookups','Refresh Window ROUTINE, after lookups')
#CLASS('Procedure Exit','Before Leaving the Procedure')
#PROMPT('&Parameters:',@s255), %Parameters
#ENABLE(%ProcedureType='FUNCTION')
 #PROMPT('Return Value:',FIELD),%ReturnValue
#ENDENABLE
#PROMPT('Window Operation Mode:',DROP('Use WINDOW setting|Normal|MDI|Modal')) %|
 ,%WindowOperationMode
#ENABLE(%INIActive)
 #BOXED('INI File Settings')
 #PROMPT('Save and Restore Window Location',CHECK) %|
 ,%INISaveWindow,DEFAULT(1),AT(10,,150)
 #ENDBOXED
#ENDENABLE
#AT(%CustomGlobalDeclarations)
 #INSERT(%StandardGlobalSetup)
#ENDAT
#INSERT(%StandardWindowCode)
```

This code starts with the `#PROCEDURE` statement, which names the Procedure template and indicates that it will have a `WINDOW` (or `APPLICATION`) structure, but no `REPORT`. The `#LOCALDATA` section defines six local variables that generate automatically as part of the procedure. These are common local variables for most generated procedures.

The `#CLASS` statements define the formula classes for the Formula Editor. These identify the source code positions at which formulas generate.

The `#PROMPT` statements create the prompts on the **Procedure Properties** window. The first allows the programmer to name the parameters passed into the procedure. The `#ENABLE` structure enables its `#PROMPT` only when the `%ProcedureType` symbol contains "FUNCTION." This occurs only when the **Prototype** prompt (standard on all procedures) contains a procedure prototype with a return data type.

The next `#PROMPT` allows the programmer to override the window's operation mode as specified on the `WINDOW` structure. The next `#ENABLE` structure enables its `#BOXED #PROMPT` only when the `%INIActive` symbol contains a value. This symbol comes from a check box on the Global Settings window.

The `#AT` structure calls the `%StandardGlobalSetup #GROUP`. This contains code to determine if the procedure is using any `.VBX` controls. If so, they are added to the list of files to ship with the application that generates into the `ProgramName.SHP` file.

You will note that none of these statements generates any target language (Clarion) source code other than the six variable declarations. The last `#INSERT` statement places all the code the `%StandardWindowCode #GROUP` generates at the end of these statements. This is the `#GROUP` that handles all the source generation for the template.

## **%StandardWindowCode #GROUP**

---

This `#GROUP` actually generates all the source code for the Window template. This includes all the local data declarations, standard window handling code, and provides all the "hooks" for all the control and extension templates to attach into the generated procedure.

```
#GROUP(%StandardWindowCode)
#IF(NOT %Window)
 #ERROR(%Procedure & ' Error: No Window Defined!')
 #RETURN
#ENDIF
#DECLARE(%FirstField)
#DECLARE(%LastField)
#DECLARE(%ProgressWindowRequired)
#INSERT(%FieldTemplateStandardButtonMenuPrompt)
#INSERT(%FieldTemplateStandardEntryPrompt)
#INSERT(%FieldTemplateStandardCheckBoxPrompt)
```

```

#EMBED(%GatherSymbols,'Gather Template Symbols'),HIDE
#INSERT(%FileControlInitialize)
%Procedure %ProcedureType%Parameters

#FOR(%LocalData)
%[20]LocalData %LocalDataStatement
#ENDFOR
#INSERT(%StandardWindowGeneration)
#IF(%ProgressWindowRequired)
#INSERT(%StandardProgressWindow)
#ENDIF
CODE
PUSHBIND
#EMBED(%ProcedureInitialize,'Initialize the Procedure')
LocalRequest = GlobalRequest
OriginalRequest = GlobalRequest
LocalResponse = RequestCancelled
ForceRefresh = False
CLEAR(GlobalRequest)
CLEAR(GlobalResponse)
#EMBED(%ProcedureSetup,'Procedure Setup')
IF KEYCODE() = MouseRight
 SETKEYCODE(0)
END
#INSERT(%StandardFormula,'Procedure Setup')
#INSERT(%FileControlOpen)
#INSERT(%StandardWindowOpening)
#EMBED(%PrepareAlerts,'Preparing Window Alerts')
#EMBED(%BeforeAccept,'Preparing to Process the Window')
#MESSAGE('Accept Handling',3)
ACCEPT
 #EMBED(%AcceptLoopBeforeEventHandling,'Accept Loop, Before CASE EVENT() handling')
 CASE EVENT()
 #EMBED(%EventCaseBeforeGenerated,'CASE EVENT() structure, before generated code')
 #INSERT(%StandardWindowHandling)
 #EMBED(%EventCaseAfterGenerated,'CASE EVENT() structure, after generated code')
 END
 #EMBED(%AcceptLoopAfterEventHandling,'Accept Loop, After CASE EVENT() handling')
 #SUSPEND
 #?CASE ACCEPTED()
 #INSERT(%StandardAcceptedHandling)
 #?END
 #RESUME
 #EMBED(%AcceptLoopBeforeFieldHandling,'Accept Loop, Before CASE FIELD() handling')
 #SUSPEND
 #?CASE FIELD()
 #EMBED(%FieldCaseBeforeGenerated,'CASE FIELD() structure, before generated code')
 #INSERT(%StandardControlHandling)
 #EMBED(%FieldCaseAfterGenerated,'CASE FIELD() structure, after generated code')
 #?END
 #RESUME
 #EMBED(%AcceptLoopAfterFieldHandling,'Accept Loop, After CASE FIELD() handling')
END
DO ProcedureReturn
!-----
ProcedureReturn ROUTINE
!|
!| This routine provides a common procedure exit point for all template
!| generated procedures.

```

```

!|
!| First, all of the files opened by this procedure are closed.
!|
!| Next, if it was opened by this procedure, the window is closed.
!|
!| Next, GlobalResponse is assigned a value to signal the calling procedure
!| what happened in this procedure.
!|
!| Next, we replace the BINDings that were in place when the procedure initialized
!| (and saved with PUSHBIND) using POPBIND.
!|
!| #IF(%ReturnValue)
!| Finally, we return to the calling procedure, passing %ReturnValue back.
#ELSE
!| Finally, we return to the calling procedure.
#ENDIF
!|
#INSERT(%FileControlClose)
#INSERT(%StandardWindowClosing)
#EMBED(%EndOfProcedure,'End of Procedure')
#INSERT(%StandardFormula,'Procedure Exit')
IF LocalResponse
 GlobalResponse = LocalResponse
ELSE
 GlobalResponse = RequestCancelled
END
POPBIND
#IF(%ProcedureType='FUNCTION')
RETURN(%ReturnValue)
#ELSE
RETURN
#ENDIF
!-----
InitializeWindow ROUTINE
!|
!| This routine is used to prepare any control templates for use. It should be called
!| once per procedure.
!|
#EMBED(%WindowInitializationCode,'Window Initialization Code')
DO RefreshWindow
!-----
RefreshWindow ROUTINE
!|
!| This routine is used to keep all displays and control templates current.
!|
IF %Window{Prop:AcceptAll} THEN EXIT.
#EMBED(%RefreshWindowBeforeLookup,'Refresh Window routine, before lookups')
#INSERT(%StandardFormula,'Before Lookups')
#INSERT(%StandardSecondaryLookups)
#INSERT(%StandardFormula,'After Lookups')
#EMBED(%RefreshWindowAfterLookup,'Refresh Window routine, after lookups')
#EMBED(%RefreshWindowBeforeDisplay,'Refresh Window routine, before DISPLAY()')
DISPLAY()
ForceRefresh = False
!-----
SyncWindow ROUTINE
#EMBED(%SyncWindowBeforeLookup,'Sync Record routine, before lookups')
#INSERT(%StandardFormula,'Before Lookups')
#INSERT(%StandardSecondaryLookups)

```

```
#INSERT(%StandardFormula,'After Lookups')
#EMBED(%SyncWindowAfterLookup,'Sync Record routine, after lookups')
!-----
#EMBED(%ProcedureRoutines,'Procedure Routines')
#EMBED(%LocalProcedures,'Local Procedures'),HIDE
```

This starts with the required `#GROUP` statement which identifies the group for use in `#INSERT` statements.

The `#IF(NOT %Window)` error check detects whether the programmer has forgotten to create a window for the procedure. The `#ERROR` statement alerts the programmer to the mistake and `#RETURN` immediately aborts any further source generation for the procedure. The `#DECLARE` statements declare two template symbols for internal use by other `#GROUPs` that are called to generate source for the procedure, and a “flag” that determines whether a “progress” window is required by the procedure.

The next three `#INSERT` statements insert `#GROUPs` that contain `#FIELD` structures to define the standard prompts that appear on the **Actions** tab for `BUTTON`, `ENTRY` and `CHECK` controls placed on the procedure’s window. These prompts allow the programmer to specify the standard actions these controls can take from this procedure.

The `%GatherSymbols #EMBED` statement has the `HIDE` attribute. This means that it will not appear in the list of available embed points for the programmer to insert code, making the embed point only available for internal use (for Code, Control, or Extension templates to generate code into).

The `#INSERT(%FileControlInitialize)` statement inserts a `#GROUP` that updates the symbols that keep track of the files used in the application with the files used by this procedure.

The `%Procedure %ProcedureType%Parameters` statement generates the first Clarion language source code for the procedure. It generates the procedure’s `PROCEDURE` statement, with or without a parameter list, as appropriate.

The `#FOR(%LocalData)` loop generates all the local variable declarations for the procedure. The  `%[20]LocalData` syntax means that the `%LocalData` symbol expands to fill at least 20 spaces before the `%LocalDataStatement` symbol expands. This aligns the data types for each variable declaration starting in column 22.

The `#INSERT(%StandardWindowGeneration)` statement generates the procedure’s `WINDOW` or `APPLICATION` data structure. This `#GROUP` also contains two `#EMBED` statements that allow the programmer to embed code either before or after the window structure.

The `#IF(%ProgressWindowRequired)` statement conditionally `#INSERTs` the `%StandardProgressWindow` group, which generates the `ProgressWindow WINDOW` structure for the procedure.

Next, the `CODE` statement generates, to begin the procedure's executable code section, which starts with a `PUSHBIND` statement to eliminate any `BIND` scoping problems. The `%ProcedureInitialize #EMBED` statement is the first programmer-available embed point in the executable code portion of the procedure.

The next six Clarion language statements are directly generated into the procedure to set it up for the action it should perform, as signaled to the procedure through the `GlobalRequest` variable. The `%ProcedureSetup #EMBED` statement is the next programmer-available embed point in the executable code portion of the procedure.

The `IF KEYCODE() = MouseRight` structure detects when the procedure has been called as a result of a `RIGHT-CLICK` popup menu. If so, it ensures that the keycode is cleared to prevent multiple execution.

The `%StandardFormula #INSERT` statement generates all the "Procedure Setup" class formulas. Following that, the `#INSERT(%FileControlOpen)` generates the code to open all the files used in the procedure (if they are not already open). This `#GROUP` also contains two `#EMBED` statements that allow the programmer to embed code either before or after the files are opened.

The `#INSERT(%StandardWindowOpening)` generates the `OPEN(window)` statement, and the `.INI` file handling code (if the programmer has checked the **Use .INI file to save and restore program settings** box). This `#GROUP` also includes two `#EMBED` statements that allow the programmer to embed code either before or after opening the window.

The next two `#EMBED` statements allow the programmer to embed code before entering the procedure's `ACCEPT` loop. `#MESSAGE` displays its message during source generation.

The `ACCEPT` loop is Clarion's event handling structure. The next `#EMBED (%AcceptLoopBeforeEventHandling)` allows the programmer to add code that will be the first to "see" any event that `ACCEPT` passes on. The `CASE EVENT()` structure contains all the code to handle field-independent events, generated from the `#INSERT(%StandardWindowHandling)` statement. This `#GROUP` is discussed in detail in its own section. The two `#EMBED` statements that surround this `#INSERT` and the one following the `CASE EVENT` structure all give the programmer the opportunity to explicitly handle any field-independent event not covered by the generated code.

The `#SUSPEND` statement means that conditional code statements (those prefaced with `#?`) will only generate if an explicit code statement (without `#?`) is also generated for the event, or if the programmer has placed some

embedded source or used a Code template in an embed point associated with the event being processed. This is the mechanism that allows Clarion's Template Language to only generate code that is actually required for the procedure, eliminating unnecessary generated code.

The `#?CASE ACCEPTED()` structure contains all the code to handle all the Accepted events for menu items. Since menu items only generate Accepted events, this structure keeps the following `CASE FIELD()` structure from becoming unwieldy. This line of code, since it is prefaced with `#?`, will only generate if there is some other code generated within it, eliminating an empty `CASE` structure. The code for the `CASE` structure is generated by the `#INSERT(%StandardAcceptedHandling)` statement. This `#GROUP` is also discussed in detail in its own section. The `#?END` statement will only generate an `END` statement if other code has already been generated.

The `#RESUME` statement terminates the `#SUSPEND` section. If no source code has actually been generated, none of the conditional source statements (prefaced by `#?`) between the `#SUSPEND` and the `#RESUME` generate.

The `#?CASE FIELD()` structure (also bracketed within `#SUSPEND` and `#RESUME` statements) contains all the code to handle all the field-specific events. The code for the `CASE` structure is generated by the `#INSERT(%StandardControlHandling)` statement (between its two `#EMBED` statements). This `#GROUP` is also discussed in detail in its own section. The `#?END` statement will only generate an `END` statement if other code has already been generated. The `#EMBED` immediately following `#RESUME` provides an embed point at the bottom of the `ACCEPT` loop.

The `END` statement terminates the `ACCEPT` loop. This statement is always generated (as is the `ACCEPT`) because every window requires an `ACCEPT` loop directly associated with it to process the events for that window. The `DO ProcedureReturn` statement calls the "cleanup code" for the procedure.

The `ProcedureReturn ROUTINE` begins with a comment block that generates into the Clarion code to explain the `ROUTINE`'s purpose. The first line of code in the `ROUTINE` is the `#INSERT(%FileControlClose)` statement. This generates the code to close the files that were opened by the procedure. This `#GROUP` also contains two `#EMBED` statements that allow the programmer to embed code either before or after the files are closed.

The `#INSERT(%StandardWindowClosing)` generates the `CLOSE(window)` statement, and the `.INI` file handling code (if the programmer has checked the **Use .INI file to save and restore program settings** box). This `#GROUP` also includes two `#EMBED` statements that allow the programmer to embed code either before or after closing the window.

The next `#EMBED` allows the programmer to embed code before closing the procedure's window. The next `#INSERT` statement generates all the "Procedure Exit" class formulas. The next five Clarion language statements set up the procedure to alert the calling procedure to the action it performed,

signaled back to the calling procedure through the GlobalRequest variable. The POPBIND statement eliminates any BIND scoping problems. The #IF structure then determines whether the procedure returns a value and generates the correct RETURN statement.

The InitializeWindow ROUTINE is a standard routine in all of Clarion's shipping Templates. It starts with a comment block, then the #EMBED allows the programmer to perform any initialization code for themselves, and provides Code, Control, and Extension templates a place to generate their window initialization code. The DO RefreshWindow statement calls the routine to display the current contents of all the controls' USE variables at the time the window is initialized.

The RefreshWindow ROUTINE is another standard routine in all of Clarion's shipping Templates that performs the procedure's MANY:1 lookups and refreshes the screen to ensure any changed data correctly displays to the user at all times. The ROUTINE starts with a comment block then the IF %Window{PROP:AcceptAll} THEN EXIT. statement. This detects when the procedure is on "non-stop" mode performing all data validity checks prior to writing a record to disk, and aborts the re-display.

The first #EMBED allows the programmer to embed code before the lookups. The next #INSERT generates all the "Before Lookups" class formulas, then #INSERT(%StandardSecondaryLookups) generates the code to get all the related records for the procedure. The next #INSERT generates all the "After Lookups" class formulas, then comes a #EMBED to allow the programmer to embed code after the lookups. The DISPLAY statement puts any changed values on screen, and ForceRefresh = False turns off the procedure's screen refresh flag.

The SyncWindow ROUTINE is also a standard routine in all of Clarion's shipping Templates. It performs the same lookups as the RefreshWindow ROUTINE, with similar embed points, but does not refresh the screen. Instead, it ensures all record buffers contain correct data. This ROUTINE is usually called before executing some action that may require the currently highlighted record in a LIST.

The next to last #EMBED statement allows the programmer to embed any ROUTINES they have called from their code within other embed points. The last #EMBED statement allows any other templates to embed any local PROCEDURES that are called from their code.

## %StandardWindowHandling #GROUP

This #GROUP generates all the code to handle field-independent events for the procedure. It generates its code inside the Window template's CASE EVENT() structure.

```
#GROUP(%StandardWindowHandling)
#FOR(%WindowEvent)
 #SUSPEND
 #?OF EVENT:%WindowEvent
 #EMBED(%WindowEventHandling,'Window Event Handling'),%WindowEvent
 #CASE(%WindowEvent)
 #OF('OpenWindow')
 IF NOT WindowInitialized
 DO InitializeWindow
 WindowInitialized = True
 END
 #IF(%FirstField)
 SELECT(%FirstField)
 #ENDIF
 #OF('GainFocus')
 ForceRefresh = True
 IF NOT WindowInitialized
 DO InitializeWindow
 WindowInitialized = True
 ELSE
 DO RefreshWindow
 END
 #OF('Sized')
 ForceRefresh = True
 DO RefreshWindow
 #ENDCASE
 #EMBED(%PostWindowEventHandling,'Window Event Handling, after generated code') %|
 ,%WindowEvent
 #RESUME
#ENDFOR
OF Event:Rejected
 #EMBED(%WindowEventHandlingBeforeRejected,'Window Event Handling - Before Rejected')
 BEEP
 DISPLAY(?)
 SELECT(?)
 #EMBED(%WindowEventHandlingAfterRejected,'Window Event Handling - After Rejected')
#SUSPEND
#?ELSE
 #EMBED(%WindowOtherEventHandling,'Other Window Event Handling')
#RESUME
```

This #GROUP starts with #FOR(%WindowEvent). This means it will loop through every instance of the %WindowEvent symbol, generating code (if required) for each field-independent event in the procedure.

The #SUSPEND statement begins the section of code that will only conditionally generate code if an explicit code statement (without #?) generates, or if the programmer has placed some embedded source or used a Code or Extension template to generate code into an embed point.

The `#?OF EVENT:%WindowEvent` statement conditionally generates an `OF` clause to the `CASE EVENT()` structure for the currently processing instance of `%WindowEvent`. This line of code, since it is prefaced with `#?`, will only generate if there is some other code generated within it, eliminating an empty `OF` clause.

The `#EMBED` statement is the key to the source generation process, and to the Procedure template's interaction with Code, Control, and Extension templates. Because it has the `“,%WindowEvent”` appended to the end, the programmer will have a separate embed point available for every instance of the `%WindowEvent` symbol. This means programmers can write their own code for any field-independent event. It also means any Code, Ccontrol, or Eextension templates the programmer places in the procedure can generate code into these embed points, as needed, to produce the code necessary to support their functionality. These embed points are the targets of the `#AT` statements used in the Code, Control, and Extension templates.

The `#CASE(%WindowEvent)` structure generates explicit source code for the field-independent events in its structure. The `#OF('OpenWindow')` checks for `EVENT:OpenWindow` and generates the check on the `WindowInitialized` variable to conditionally initialize the window and set `WindowInitialized` to true. This code executes if no `EVENT:GainFocus` has already occurred (such as opening a second window on the same execution thread that currently has focus). The `SELECT(%FirstField)` statement is generated only if there are any controls that can receive focus in the window.

The `#OF('GainFocus')` statement checks for `EVENT:GainFocus` and generates the `ForceRefresh = True`, then checks to see if the window has already been initialized (if the user is switching between active threads it would have been). If not, it initializes the window, otherwise it simply refreshes it. The `#OF('Sized')` statement checks for `EVENT:Sized` and generates `ForceRefresh = True` and `DO RefreshWindow` to refresh the window after the user has resized it.

The `#ENDCASE` statement terminates the `#CASE` structure. The next `#EMBED` allows the programmer an opportunity to embed their own code following the generated code for any of these events. The `#RESUME` statement terminates the `#SUSPEND` section. If no source code has actually been generated, no conditional source statements (prefaced by `#?`) between the `#SUSPEND` and the `#RESUME` are generated.

`#ENDFOR` terminates the `#FOR` loop, then `OF EVENT:Rejected` generates and offers before and after embed points surrounding standard code to alert the user that the data input into the current control has been rejected (usually the data is out of range) and leave them on the same control.

The `#SUSPEND` statement begins another conditional generation section. This means the `#?ELSE` statement only generates an `ELSE` if source code is generated by the `#EMBED` statement. `#RESUME` terminates this `#SUSPEND` section.

## %StandardAcceptedHandling #GROUP

This #GROUP generates all the code to handle field-specific events for the procedure. It generates its code inside the Window Template's CASE FIELD() structure.

```
#GROUP(%StandardAcceptedHandling)
#FOR(%Control),WHERE(%ControlMenu)
 #FIX(%ControlEvent,'Accepted')
 #MESSAGE('Control Handling: ' & %Control,3)
 #SUSPEND
#?OF %Control
 #EMBED(%ControlPreEventHandling,'Control Event Handling, before generated code') %|
 ,%Control,%ControlEvent

 #INSERT(%FieldTemplateStandardHandling)
 #EMBED(%ControlEventHandling,'Internal Control Event Handling') %|
 ,%Control,%ControlEvent,HIDE

 #EMBED(%ControlPostEventHandling,'Control Event Handling, after generated code') %|
 ,%Control,%ControlEvent

#RESUME
#ENDFOR
```

This code starts with the #FOR(%Control),WHERE(%ControlMenu) statement. The WHERE attribute limits this #FOR loop to only those instances of %Control that contain menu items. The #FIX statement ensure that this code only deals with Accepted events.

The #MESSAGE statement displays its message during source generation. #SUSPEND begins a conditional source generation section.

The #?OF %Control statement conditionally generates an OF clause to the CASE ACCEPTED() structure for the currently processing instance of %Control. This line of code, since it is prefaced with #?, will only generate if there is some other code generated within it, eliminating an empty OF clause.

All three #EMBED statements have “,%Control,%ControlEvent” appended to the end, so the programmer will have a separate embed point available for every instance of the %ControlEvent symbol within every instance of the %Control symbol. For this group, this only means the Accepted event.

The #INSERT(%FieldTemplateStandardHandling) statement generates code to handle all the Actions dialog selections the programmer has made for the menu item. The next two #EMBED statements also have “,%Control,%ControlEvent” appended to the end. The first has the HIDE attribute, so it is available only for Code, Control, and Extension template use. These three #EMBEDs give the programmer an embed point both before and after any code automatically generated for them by the Actions tab prompts.

#RESUME terminates this #SUSPEND section. #ENDFOR terminates the %Control loop.

## %StandardControlHandling #GROUP

This #GROUP generates all the code to handle field-specific events for the procedure. It generates its code inside the Window template's CASE FIELD() structure.

```
#GROUP(%StandardControlHandling)
#FOR(%Control),WHERE(%Control)
 #MESSAGE('Control Handling: ' & %Control,3)
 #SUSPEND
#?OF %Control
 #EMBED(%ControlPreEventCaseHandling,'Control Handling, before event handling') %|
 ,%Control

 #?CASE EVENT()
 #IF(NOT %ControlMenu)
 #FOR(%ControlEvent)
 #SUSPEND
 #?OF EVENT:%ControlEvent
 #EMBED(%ControlPreEventHandling,'Control Event Handling, Before Generated %|
 Code'),%Control,%ControlEvent

 #INSERT(%FieldTemplateStandardHandling)
 #EMBED(%ControlEventHandling,'Internal Control Event Handling') %|
 ,%Control,%ControlEvent,HIDE
 #EMBED(%ControlPostEventHandling,'Control Event Handling, After Generated %|
 Code'),%Control,%ControlEvent

 #RESUME
 #ENDFOR
 #ELSE
#?OF EVENT:Accepted
 #ENDIF
 #SUSPEND
#?ELSE
 #EMBED(%ControlOtherEventHandling,'Other Control Event Handling'),%Control
#RESUME
#?END
#EMBED(%ControlPostEventCaseHandling,'Control Handling, after event handling') %|
 ,%Control

#RESUME
#ENDFOR
```

This code starts with the #FOR(%Control),WHERE(%Control) statement. The WHERE clause may at first seem redundant, since #FOR will only loop through existing instances of %Control. However, since some controls do not need (and so do not have) field equate labels, there are valid instances of %Control that do not contain a value for %Control itself. Therefore, the WHERE attribute limits this #FOR loop to those instances of %Control that do contain a field equate label for the control.

The #MESSAGE statement displays its message during source generation. #SUSPEND begins a conditional source generation section.

The #?OF %Control statement conditionally generates an OF clause to the CASE FIELD() structure for the currently processing instance of %Control. This line of code, since it is prefaced with #?, will only generate if there is some other code generated within it, eliminating an empty OF clause.

The first `#EMBED` allows the programmer to handle any situation that needs to be handled before any generated code for the control. The `#!CASE EVENT()` conditionally generates a `CASE EVENT()` structure for the control. The `#!IF(NOT %ControlMenu)` statement filters out all the menu items, since they are handled by the `%StandardAcceptedHandling #GROUP`. `#!FOR(%ControlEvent)` loops through all the possible events that the control being processed can generate.

`#!SUSPEND` begins another conditional source generation section, nested within the previous one. This allows multiple levels of conditional source code generation. The outer section is automatically generated if any code is generated from the inner section.

The `#!OF EVENT:%ControlEvent` statement conditionally generates an `OF` clause to the `CASE EVENT()` structure for the currently processing instance of `%Control`. This line of code, since it is prefaced with `#!`, will only generate if there is some other code generated within it, eliminating an empty `OF` clause.

This next `#EMBED` statement has “`,%Control,%ControlEvent`” appended to the end, so the programmer will have a separate embed point available for every instance of the `%ControlEvent` symbol within every instance of the `%Control` symbol. This means programmers can write their own code for any field-specific event, for any control. It also means any Code templates, Control templates, or Extension templates the programmer places in the procedure can generate code into these embed points, as needed, to produce the code necessary to support their functionality. These embed points are the targets of the `#!AT` statements used in the Code, Control, and Extension templates.

The `#!INSERT(%FieldTemplateStandardHandling)` statement generates code to handle all the Actions tab selections the programmer has made for the control. The prompts on the Actions tab come from the `#!FIELD` structures that were `#!INSERTed` at the beginning of the Window template.

The next two `#EMBED` statements also have “`,%Control,%ControlEvent`” appended to the end, so the programmer will have a separate embed point available for every instance of the `%ControlEvent` symbol within every instance of the `%Control` symbol. The first has the `HIDE` attribute, so it is available only for Code, Control, and Extension template use. These three `#EMBEDs` give the programmer an embed point both before and after any code automatically generated for them by the Actions tab prompts.

`#!RESUME` terminates the inner conditional source generation section, then `#!ENDFOR` terminates the `%ControlEvent` loop. The `#!ELSE` refers back to the `#!IF(NOT %ControlMenu)` and will generate an empty `OF EVENT:Accepted` followed by an `ELSE` statement for a Menu item if the programmer has entered code into the Other Control Event Handling embed point. This eliminates any duplication between `EVENT:Accepted` code for a

menu item while still allowing the programmer to process any user-defined events for them.

The #SUSPEND statement begins another nested conditional generation section. This means the #?ELSE statement only generates an ELSE if source code is generated by the #EMBED statement. #RESUME terminates this #SUSPEND section.

The #?END generates the END statement for the CASE FIELD() structure, if any code has been generated, then the #RESUME statement terminates the outer #SUSPEND section. #ENDFOR terminates the %Control loop.

## Code Template: ControlValueValidation

The *ControlValueValidation* Code template performs data entry validation for an entry-type control (ENTRY, SPIN, or COMBO) by looking up the value entered by the user in another data file. If the lookup is successful, the entered value is valid. If not, it calls another procedure to allow the user to select a valid value from the lookup file. This Code template is designed to generate code only into EVENT:Selected or EVENT:Accepted embed points of an ENTRY, SPIN, or COMBO control. These are the controls into which a user can directly type in data.

```
#CODE(ControlValueValidation,'Control Value Validation')
#RESTRICT
 #CASE(%ControlType)
 #OF('ENTRY')
 #OROF('SPIN')
 #OROF('COMBO')
 #CASE(%ControlEvent)
 #OF('Accepted')
 #OROF('Selected')
 #ACCEPT
 #ELSE
 #REJECT
 #ENDCASE
#ELSE
 #REJECT
#ENDCASE
#ENDRESTRICT
#DISPLAY('This Code Template is used to perform a control value')
#DISPLAY('validation. This Code Template only works for')
#DISPLAY('the Selected or Accepted Events for an Entry Control.')
#DISPLAY('')
#PROMPT('Lookup Key',KEY),%LookupKey,REQ
#PROMPT('Lookup Field',COMPONENT),%LookupField,REQ
#PROMPT('Lookup Procedure',PROCEDURE),%LookupProcedure
#DISPLAY('')
#DISPLAY('The Lookup Key is the key used to perform the value validation.')
#DISPLAY('If the Lookup Key is a multi-component key, you must insure that')
#DISPLAY('other key elements are primed BEFORE this Code Template is used.')
#DISPLAY('')
#DISPLAY('The Lookup field must be a component of the Lookup Key. Before')
#DISPLAY('execution of the lookup code, this field will be assigned the value of')
#DISPLAY('the control being validated, and the control will be assigned the value')
#DISPLAY('of the lookup field if the Lookup procedure is successful.')
#DISPLAY('')
#DISPLAY('The Lookup Procedure is called to let the user to select a value. ')
#DISPLAY('Request upon entrance to the Lookup will be set to SelectRecord, and ')
#DISPLAY('successful completion is signalled when Response = RequestCompleted.')
#IF(%ControlEvent='Accepted')
 IF %Control{PROP:Req} = False AND NOT %ControlUse #<! If not required and empty
ELSE
 #INSERT(%CodeTPLValidationCode)
END
#ELIF(%ControlEvent='Selected')
 #INSERT(%CodeTPLValidationCode)
#ELSE
 #ERROR('This Code Template must be used for Accepted or Selected Events!')
#ENDIF
```

A Code template always starts with the #CODE statement, which identifies it within the template set and defines the description which appears in the list of available Code templates for a given embed point.

The #RESTRICT structure defines the embed points where the code template will appear as a choice. The #CASE(%ControlType) structure limits the embed points to the ENTRY, SPIN, and COMBO controls, and the #CASE(%ControlEvent) structure limits the embed points to EVENT:Accepted and EVENT:Selected.

The #ACCEPT statement indicates these are appropriate embed points, while the #REJECT indicates all other control type and event embed points are not valid for the Code template to appear in as a choice.

All the #DISPLAY statements display their text to the programmer on the code template's prompt dialog. These describe the information the programmer needs to supply in the prompts.

The first #PROMPT asks for the name of the key to use in the file that will be used to validate the user's input. The REQ attribute indicates the programmer must supply this information.

The second #PROMPT asks for the name of the field in the key that contains the same information the user should enter into the control. Again, the REQ attribute indicates the programmer must supply this information.

The third #PROMPT asks for the name of the procedure to call if the lookup is unsuccessful. This would usually be a Browse procedure for the lookup file with a Select button to allow the user to choose the record containing the value they want for the control.

Again, the #DISPLAY statements display text to the programmer on the prompt dialog to describe the information the programmer needs to supply in the prompts.

The #IF(%ControlEvent='Accepted') structure generates an IF structure for EVENT:Accepted that detects when the control has the REQ attribute or the user has entered a value and #INSERTs the %CodeTPLValidationCode #GROUP to generate the source code for the data validation. The #ELSIF just unconditionally #INSERTs the %CodeTPLValidationCode #GROUP to generate the source code for the data validation.

If the event is anything other than EVENT:Accepted or EVENT:Selected, an error message is the only output generated.

## %CodeTPLValidationCode #GROUP

This #GROUP is the “workhorse” of the Code template. It generates the actual file lookup code to validate the data entry. It takes the information provided in the prompts and combines it with the %ControlUse symbol to generate a GET statement into the lookup file. If the GET is successful, the data is valid. If not, it calls the lookup procedure.

```
#GROUP(%CodeTPLValidationCode)
 %LookupField = %ControlUse
#FIND(%Field,%LookupField)
 GET(%File,%LookupKey)
 IF ERRORCODE()
 GlobalRequest = SelectRecord
 %LookupProcedure
 LocalResponse = GlobalResponse
 GlobalResponse = RequestCancelled
 IF LocalResponse = RequestCompleted
 %ControlUse = %LookupField
#IF(%ControlEvent='Accepted')
 ELSE
 SELECT(%Control)
 CYCLE
#ENDIF
 END
#IF(%ControlEvent='Selected')
 SELECT(%Control)
#ENDIF
 END
```

```
#<! Move value for lookup
#! FIX field for lookup
#<! Get value from file
#<! IF record not found
#<! Set Action for Lookup
#<! Call Lookup Procedure
#<! Save Returned Action
#<! Clear the Action Value
#<! IF Lookup successful
#<! Move value to control field
#! IF a Post-Edit Validation
#<! ELSE (IF Lookup NOT...)
#<! Select the control
#<! Go back to ACCEPT
#! END (IF a Pre-Edit...)
#<! END (IF Lookup successful)
#! IF a Pre-Edit Validation
#<! Select the control
#! END (IF a Pre-Edit...)
#<! END (IF record not found)
```

This #GROUP starts by generating the %LookupField = %ControlUse assignment. This assigns the control’s USE variable to the field named in the second prompt; the key field that should contain the correct value.

The #FIND(%Field,%LookupField) statement looks through all the fields in the data dictionary, looking for a matching field to the one contained in %LookupField. This fixes %Field and %File to the correct values to generate the GET(%File,%LookupKey) statement. This becomes a GET(file,key) form of the GET statement to get a single record from the lookup file with matching key field values.

The IF ERRORCODE() structure checks for a successful GET operation. If an error occurred, the GET was unsuccessful and the GlobalRequest = SelectRecord statement sets up the call to the lookup procedure, generated by the %LookupProcedure statement.

After return from the lookup procedure, LocalResponse = GlobalResponse saves the lookup procedure’s response code. Then the GlobalResponse = RequestCancelled statement cleans up so any other execution thread does not get an incorrect response. This must be done immediately, before the user has a chance to change execution threads.

The IF LocalResponse = RequestCompleted structure detects a user choice from the lookup procedure and the %ControlUse = %LookupField statement assigns the choice to the control's USE variable.

The #IF(%ControlEvent='Accepted') detects when the Code template is generating for EVENT:Accepted and adds the ELSE clause to SELECT the control and CYCLE back to the top of the ACCEPT loop.

The END statement terminates the IF LocalResponse = RequestCompleted structure. The #IF(%ControlEvent='Selected') structure generates the SELECT statement for the control when generating for EVENT:Selected.

The END statement terminates the IF ERRORCODE() structure. Obviously, if there was no error on the GET statement, the data is valid and no further code is necessary.

## Control Template: *DOSFileLookup*

The *DOSFileLookup* Control template adds an ellipsis (...) button which leads the end user to a standard **Open File** dialog. You can specify a file mask, and a return variable to hold the end user's choice.

```
#CONTROL(DOSFileLookup,'Lookup a DOS file name'),WINDOW,MULTI
CONTROLS
 BUTTON('...'),AT(, ,12,12),USE(?LookupFile)
END
#BOXED('DOS File Lookup Prompts')
#PROMPT('&File Dialog Header:',@S60),%DOSFileDialogHeader,REQ,DEFAULT('Choose File')
#PROMPT('&DOS FileName Variable:',FIELD),%DOSFileField,REQ
#PROMPT('Default Directory:',@S80),%DOSInitialDirectory
#PROMPT('&Return to original directory when done.',CHECK),%ReturnToOriginalDir,AT(10)
#PROMPT('&Use a variable to specify the file mask(s).',CHECK),%DOSVariableMask,AT(10)
#ENABLE(%DOSVariableMask)
 #PROMPT('Variable Mask Value:',FIELD),%DOSVariableMaskValue
#ENDENABLE
#ENABLE(NOT %DOSVariableMask)
 #PROMPT('File Mask Description:',@S40),%DOSMaskDesc,REQ,DEFAULT('All Files')
 #PROMPT('File Mask',@S50),%DOSMask,REQ,DEFAULT('*.*')
 #BUTTON('More File Masks'),MULTI(%DOSMoreMasks,%DOSMoreMaskDesc&'-'&%DOSMoreMask)
 #PROMPT('File Mask Description:',@S40),%DOSMoreMaskDesc,REQ
 #PROMPT('File Mask',@S50),%DOSMoreMask,REQ
 #ENDBUTTON
#ENDENABLE
#ENDBOXED
#LOCALDATA
DOSDialogHeader CSTRING(40)
DOSExtParameter CSTRING(250)
DOSTargetVariable CSTRING(80)
#ENDLOCALDATA
#ATSTART
 #DECLARE(%DOSExtensionParameter)
 #DECLARE(%DOSLookupControl)
 #FOR(%Control),WHERE(%ControlInstance = %ActiveTemplateInstance)
 #SET(%DOSLookupControl,%Control)
 #ENDFOR
 #IF(NOT %DOSVariableMask)
 #SET(%DOSExtensionParameter,%DOSMaskDesc & '|' & %DOSMask)
 #FOR(%DOSMoreMasks)
 #SET(%DOSExtensionParameter,%DOSExtensionParameter & '|' & %DOSMoreMaskDesc &
 & '|' & %DOSMoreMask)
 #ENDFOR
 #END
#ENDAT
#AT(%ControlEventHandler,%DOSLookupControl,'Accepted')
IF NOT %DOSFileField
 #INSERT(%StandardValueAssignment,'DOSTargetVariable',%DOSInitialDirectory)
ELSE
 DOSTargetVariable = %DOSFileField
END
#INSERT(%StandardValueAssignment,'DOSDialogHeader',%DOSFileDialogHeader)
#IF(%DOSVariableMask)
DOSExtParameter = %DOSVariableMaskValue
#ELSE
DOSExtParameter = '%DOSExtensionParameter'
#ENDIF
```

```

#IF(%ReturnToOriginalDir)
IF FILEDIALOG(DOSDialogHeader,DOSTargetVariable,DOSExtParameter,FILE:KeepDIR)
#ELSE
IF FILEDIALOG(DOSDialogHeader,DOSTargetVariable,DOSExtParameter,0)
#ENDIF
 %DOSFileField = DOSTargetVariable
 DO RefreshWindow
END
#ENDAT

```

This starts, as all Control templates must, with a **#CONTROL** statement. The **WINDOW** attribute allows you to populate it onto a window, but not onto a report. The **MULTI** attribute specifies that the template may be populated multiple times onto the same window. The **CONTROLS** section pre-defines the **BUTTON** control for the window.

The **#BOXED** structure places a box around all the prompts that display on the Actions tab for this Control template. The first **#PROMPT** asks for the text for the caption of the **Open File** dialog, and the next asks for the name of a variable to receive the end user's choice. The third allows you to explicitly set the directory in which the **Open File** dialog starts.

The fourth **#PROMPT** is a check box asking whether the program should return to the directory from which it started from the **Open File** dialog. The fifth **#PROMPT** is a check box asking whether the programmer will explicitly set the file mask(s) for the **Open File** dialog, or use a variable to determine them at run time. When checked, the first **#ENABLE** activates the Variable Mask Value **#PROMPT** to get the name of the variable to use at run time. If not checked, the second **#ENABLE** activates its set of prompts to get each explicit file mask to pass to the **Open File** dialog.

The **#LOCALDATA** section defines three local variables that generate automatically as part of the procedure. These local variables are only used in the code generated by this Control template as the actual variables passed as parameters to the **FILEDIALOG** procedure.

The **#ATSTART** statement begins a section of template code that executes before any source code generates for the procedure. This means it is only appropriate to initialize user-defined template symbols and perform any necessary set up to generate correct source for the control template into the procedure. This section does not generate source code. The **#DECLARE** statements declare two symbols used only during source generation for this Control template.

**#FOR(%Control),WHERE(%ControlInstance=%ActiveTemplateInstance)** executes the enclosed **#SET** statement only for the single control populated by this Control template. The **#SET** statement then places the field equate label of the control into **%DOSLookupControl**.

The **#IF** structure checks whether the programmer checked the **Use a Variable to specify the file mask(s)** box and if not, sets up the specific file

masks the programmer chose to pass to the FILEDIALOG procedure. The #ENDAT statement terminates the #ATSTART section.

The next #AT generates Clarion code into the embed point for the Accepted event for the control populated by this Control template to perform the file lookup. The IF NOT %DOSFileField structure detects whether the user has performed the lookup. If they haven't the initial directory is assigned to the DOSTargetVariable. If the user has performed the lookup, the ELSE clause assigns the result of the previous lookup as the starting point for the next.

The #INSERT statement creates assignment statements to initialize the **File Open** dialog's title. Next, the #IF(%DOSVariableMask) conditionally generates an assignment to initialize the file mask to pass to the **File Open** dialog.

The #IF (%ReturnToOriginalDir) generates the correct IF FILEDIALOG structure performs the actual lookup for the file, either to return to the original directory or not. If the user selects a file from the **File Open** dialog, the selected filename is assigned to variable the user selected in the **DOS FileName Variable** prompt, then the DO RefreshWindow statement ensures that all data on the window is current. The #ENDAT statement terminates the #AT section.

## Extension Template: *DateTimeDisplay*

The *DateTimeDisplay* Extension template displays the date and/or time in either a display-only STRING control or a section of the status bar. Of course, the status bar should be declared on the window.

```
#EXTENSION(DateTimeDisplay,'Display the date and/or time in the current window') %|
 ,HLP('~TPLExtensionDateTimeDisplay'),PROCEDURE
#BUTTON('Date and Time Display'),AT(10,,180)
#BOXED('Date Display...')
 #PROMPT('Display the current day/date in the window',CHECK) %|
 ,%DisplayDate,DEFAULT(0),AT(10,,150)
#ENABLE(%DisplayDate)
 #PROMPT('Date Picture:',DROP('October 31, 1959|OCT 31,1959|10/31/59| %|
 10/31/1959|31 OCT 59|31 OCT 1959|31/10/59| %|
 31/10/1959|Other')),%DatePicture %|
 ,DEFAULT('October 31, 1959')
#ENABLE(%DatePicture = 'Other')
 #PROMPT('Other Date Picture:',@S20),%OtherDatePicture,REQ
#ENDENABLE
#PROMPT('Show the day of the week before the date',CHECK),%ShowDayOfWeek %|
 ,DEFAULT(1),AT(10,,150)
#PROMPT('&Location of Date Display:',DROP('Control|Status Bar')) %|
 ,%DateDisplayLocation
#ENABLE(%DateDisplayLocation='Status Bar')
 #PROMPT('Status Bar Section:',@n1),%DateStatusSection,REQ,DEFAULT(1)
#ENDENABLE
#ENABLE(%DateDisplayLocation='Control')
 #PROMPT('Date Display Control:',CONTROL),%DateControl,REQ
#ENDENABLE
#ENDENABLE
#ENDBOXED
#BOXED('Time Display...')
 #PROMPT('Display the current time in the window',CHECK),%DisplayTime %|
 ,DEFAULT(0),AT(10,,150)
#ENABLE(%DisplayTime)
 #PROMPT('Time Picture:',DROP('5:30PM|5:30:00PM|17:30|17:30:00| %|
 1730|173000|Other')),%TimePicture %|
 ,DEFAULT('5:30PM')
#ENABLE(%TimePicture = 'Other')
 #PROMPT('Other Time Picture:',@S20),%OtherTimePicture,REQ
#ENDENABLE
#PROMPT('&Location of Time Display:',DROP('Control|Status Bar')) %|
 ,%TimeDisplayLocation
#ENABLE(%TimeDisplayLocation='Status Bar')
 #PROMPT('Status Bar Section:',@n1),%TimeStatusSection,REQ,DEFAULT(2)
#ENDENABLE
#ENABLE(%TimeDisplayLocation='Control')
 #PROMPT('Time Display Control:'.CONTROL),%TimeControl,REQ
#ENDENABLE
#ENDENABLE
#ENDBOXED
#ENDBUTTON
#ATSTART
#DECLARE(%TimerEventGenerated)
#IF(%DisplayDate)
#DECLARE(%DateUsePicture)
#CASE(%DatePicture)
```

```

#OF('10/31/59')
 #SET(%DateUsePicture,'@D1')
#OF('10/31/1959')
 #SET(%DateUsePicture,'@D2')
#OF('OCT 31,1959')
 #SET(%DateUsePicture,'@D3')
#OF('October 31, 1959')
 #SET(%DateUsePicture,'@D4')
#OF('31/10/59')
 #SET(%DateUsePicture,'@D5')
#OF('31/10/1959')
 #SET(%DateUsePicture,'@D6')
#OF('31 OCT 59')
 #SET(%DateUsePicture,'@D7')
#OF('31 OCT 1959')
 #SET(%DateUsePicture,'@D8')
#OF('Other')
 #SET(%DateUsePicture,%OtherDatePicture)
#ENDCASE
#ENDIF
#IF(%DisplayTime)
#DECLARE(%TimeUsePicture)
#CASE(%TimePicture)
#OF('17:30')
 #SET(%TimeUsePicture,'@T1')
#OF('1730')
 #SET(%TimeUsePicture,'@T2')
#OF('5:30PM')
 #SET(%TimeUsePicture,'@T3')
#OF('17:30:00')
 #SET(%TimeUsePicture,'@T4')
#OF('173000')
 #SET(%TimeUsePicture,'@T5')
#OF('5:30:00PM')
 #SET(%TimeUsePicture,'@T6')
#OF('Other')
 #SET(%TimeUsePicture,%OtherTimePicture)
#ENDCASE
#ENDIF
#ENDAT
#AT(%DataSectionBeforeWindow)
 #IF(%DisplayDate AND %ShowDayOfWeek)
DisplayDayString STRING('Sunday Monday Tuesday WednesdayThursday %|
 Friday Saturday ')
DisplayDayText STRING(9),DIM(7),OVER(DisplayDayString)
 #ENDIF
#ENDAT
#AT(%BeforeAccept)
 #IF(%DisplayTime OR %DisplayDate)
IF NOT INRANGE(%Window{Prop:Timer},1,100)
 %Window{Prop:Timer} = 100
END
#INSERT(%DateTimeDisplayCode)
#ENDIF
#ENDAT
#AT(%WindowEventHandling,'Timer')
 #SET(%TimerEventGenerated,%True)
 #IF(%DisplayDate OR %DisplayTime)
#INSERT(%DateTimeDisplayCode)

```

```

#ENDIF
#ENDAT
#AT(%WindowOtherEventHandling)
#IF(%DisplayDate OR %DisplayTime)
#IF(NOT %TimerEventGenerated)
IF EVENT() = EVENT:Timer
#INSERT(%DateTimeDisplayCode)
END
#ENDIF
#ENDIF
#ENDAT

```

An Extension template starts with the `#EXTENSION` statement. The `PROCEDURE` attribute specifies the Extension template is available only at the procedure level, not the global level of the application.

The `#BUTTON` structure creates a separate page for all the prompts for this Extension template. These prompts ask the programmer for the format of the date and/or time to display, and whether to display them in a control or the status bar.

The `#ATSTART` statement begins a section of template code that executes before any source code generates for the procedure. This means it is only appropriate to initialize user-defined template symbols and perform any necessary set up to generate correct source for the control template into the procedure. This section does not generate source code.

The `#DECLARE(%TimerEventGenerated)` statement declares a symbol used only in this Extension template. It is used to flag whether an `OF EVENT:Timer` clause has been generated for the procedure.

The `#IF(%DisplayDate)` structure sets up to display the date by declaring a symbol to contain the programmer's choice of date formats. The `#CASE` structure assigns that choice to the `%DateUsePicture` symbol. The `#IF(%DisplayTime)` structure sets up to display the time by declaring a symbol to contain the programmer's choice of date formats. The `#CASE` structure assigns that choice to the `%TimeUsePicture` symbol. The `#ENDAT` statement terminates the `#ATSTART` section.

The next `#AT` generates code into the embed point that appears immediately before the window data structure. The `#IF(%DisplayDate AND %ShowDayOfWeek)` structure generates two local variable declarations for the procedure if the programmer is displaying the date with the day of week.

The next `#AT` generates code into the embed point that appears immediately before the `ACCEPT` loop. The `#IF(%DisplayTime OR %DisplayDate)` structure generates code that ensures the window has its `TIMER` attribute set. The `IF NOT INRANGE(%Window{Prop:Timer},1,100)` detects the lack of the attribute, then `%Window{Prop:Timer} = 100` sets it to one second. The `#INSERT(%DateTimeDisplayCode)` adds the code that updates the display.

The next #AT generates code into the embed point for EVENT:Timer. This embed point only appears if the programmer has placed a TIMER attribute on the window. Therefore, the #SET(%TimerEventGenerated,%True) statement signals that code was generated in this embed point. The #IF(%DisplayTime OR %DisplayDate) structure ensures the #INSERT(%DateTimeDisplayCode) statement generates the code that updates the display every time EVENT:Timer is processed.

The next #AT generates code into the “Other Window Event Handling” embed point. This is the ELSE clause of the CASE EVENT() structure to handle field-independent events. #IF(NOT %TimerEventGenerated) detects that the previous #AT did not generate code because the programmer did not place the TIMER attribute on the window. Therefore, the IF EVENT() = EVENT:Timer structure is necessary for the code that updates the display whenever EVENT:Timer occurs.

## %DateTimeDisplayCode #GROUP

---

This #GROUP generates the code to actually display the Date and/or Time.

```
#GROUP(%DateTimeDisplayCode)
 #IF(%DisplayDate)
 #IF(%ShowDayOfWeek)
 #CASE(%DateDisplayLocation)
 #OF('Control')
 %DateControl{Prop:Text} = CLIP(DisplayDayText[(TODAY()%%7)+1]) & ', ' & %|
 FORMAT(TODAY(),%DateUsePicture)
 DISPLAY(%DateControl)
 #ELSE
 %Window{Prop:StatusText,%DateStatusSection} = CLIP(DisplayDayText[(%|
 TODAY()%%7)+1]) & ', ' & FORMAT(TODAY(),%DateUsePicture)
 #ENDCASE
 #ELSE
 #CASE(%DateDisplayLocation)
 #OF('Control')
 %DateControl{Prop:Text} = FORMAT(TODAY(),%DateUsePicture)
 DISPLAY(%DateControl)
 #ELSE
 %Window{Prop:StatusText,%DateStatusSection} = FORMAT(TODAY(),%DateUsePicture)
 #ENDCASE
 #ENDIF
 #ENDIF
 #IF(%DisplayTime)
 #CASE(%TimeDisplayLocation)
 #OF('Control')
 %TimeControl{Prop:Text} = FORMAT(CLOCK(),%TimeUsePicture)
 DISPLAY(%DateControl)
 #ELSE
 %Window{Prop:StatusText,%TimeStatusSection} = FORMAT(CLOCK(),%TimeUsePicture)
 #ENDCASE
 #ENDIF
```

The #IF(%DisplayDate) structure generates the code to display the date. The #IF(%ShowDayOfWeek) structure detects the programmer's choice to display the day along with the date, then #CASE(%DateDisplayLocation) generates the code to display into a STRING display-only control for the #OF('Control') clause.

The assignment statement concatenates the day of the week (from the DisplayDayText[(TODAY() %% 7)+1] expression) with the formatted date (from the FORMAT(TODAY(),%DateUsePicture) expression) into the STRING(text) property (the %DateControl{Prop:Text} property). The %% generates as a single % (modulus operator) for the TODAY() % 7 + 1 expression to get the correct day of the week text from the DisplayDayText array. The #ELSE clause of the #CASE(%DateDisplayLocation) assigns the same expression to the status bar section the programmer chose for the date display.

The #ELSE clause of the #IF(%ShowDayOfWeek) structure performs the same assignments, without the day of the week. The #IF(%DisplayTime) structure performs the same type of assignments of the formatted time to either a STRING display-only control or the status bar.

# PART IV

---

## ISAM DATABASE DRIVERS



## 23 - DATABASE DRIVER OVERVIEW

### Data Independence

Clarion achieves database independence with its built-in driver technology, which lets you access data from virtually any file system using the same set of Clarion language commands. Many file drivers are available and more are being added.

**Tip:** Before you can use a database driver, it must be registered with the Clarion environment. The database drivers in this package are pre-registered. See the *User's Guide—Configuring the Environment* for more information on registering add-on database drivers.

The Clarion language commands for accessing data are the same regardless of the file system you use; simply choose the appropriate file driver from the drop-down list in your Data Dictionary, then don't worry about it. The file driver translates the Clarion commands to the chosen file system's native format. The drivers read and write in the file system's native format without temporary files or import/export routines.

### Choosing the Right Driver

---

Choosing a file system is an important decision, and we encourage you to gather as much information from as many good sources as you can to support your decision. Although the choice of file systems is important, with Clarion, it is not irrevocable. If the file system you choose does not live up to your expectations, you can change to one that does. For example, some developers use the TopSpeed file system for project development, then switch to an SQL file system during project implementation in order to postpone the expense of the SQL software and server hardware until late in the development cycle.

Some factors you should consider when choosing a file system include:

- data sharing requirements (how many concurrent users)
- data volume (size and number of files and records)
- data types (dates, times, memos, blobs, etc.)
- business rules (triggers, relational integrity, transaction processing)
- speed versus robustness and flexibility tradeoffs
- typical processing requirements (does your system add a few records at a time or does it regularly read or update the entire database; do your users need up-to-the-second data or yesterday's or last week's data)

## Common Driver Features

### Importing File Definitions

---

For existing data, you can generally import file definitions into your Clarion data dictionary. We strongly recommend importing file definitions whenever possible, because it reduces your development time and effort, plus it results in fewer errors in file definitions. See *The Dictionary Editor* in the *User's Guide* for more information on importing files.

**Tip:** You can use the Enterprise Edition Dictionary Synchronizer to import an entire database definition, including file relationships, in a single pass.

### Keys, Indexes, and Performance

---

Although you may define indexes within your Clarion data dictionary that do not exist within the native file system, we do not recommend doing so because your application performance will generally suffer. Instead, we recommend defining the required key or index with the native file system's tools, then importing the file definition, including the key or index definitions, into your Clarion data dictionary.

### Sorting and Collating Sequences

---

By default, all TopSpeed's database drivers sort using the ANSI collating sequence. Adding the OEM attribute causes the driver to use the ASCII collating sequence.

### Debugging and Tracing File I/O

---

All of TopSpeed's Database drivers can create a log file documenting Clarion I/O statements they process, the corresponding file system commands, and their return codes.

You can generate system-wide logs and on-demand logs (conditional logging based on your program logic).

#### **System-wide Logging**

For system-wide logging, you can add the following to your WIN.INI file:

```
[CWdriver]
Profile=[1|0]
```

```
Details=[1|0]
TraceFile=[Pathname]
```

where *driver* is the database driver name (for example [CWTopSpeed]). Neither the INI section name [CW*driver*] nor the INI entry names are case sensitive.

Profile=1 tells the driver to include the Clarion I/O statements in the log file; Profile=0 tells the driver to omit Clarion I/O statements. The Profile switch must be turned on for the Details switch to have any effect.

Details=1 tells the driver to include record buffer contents in the log file; however, if the file is encrypted, you must turn on both the Details switch and the ALLOWDETAILS switch to log record buffer contents (see *ALLOWDETAILS*). Details=0 tells the driver to omit record buffer contents. The Profile switch must be turned on for the Details switch to have any effect.

TraceFile names the log file to write to. If TraceFile is omitted the driver writes the log to *driver.log* in the current directory. *Pathname* is the full pathname or the filename of the log file to write. If no path is specified, the driver writes the specified file to the current directory.

Logging opens the named logfile for exclusive access. If the file exists, the new log data is appended to the file.

**Tip:** The Trace example program (installed by default to Clarion4\Examples\Resource\Trace) manages the .INI settings for all database drivers, as well as the Clarion view engine.

## On Demand Logging

For on-demand logging you can use property syntax within your program to conditionally turn various levels of logging on and off. The logging is effective for the target table and any view for which the target table is the primary table.

```
file{PROP:Profile}=Pathname !Turns Clarion I/O logging on
file{PROP:Profile}='' !Turns Clarion I/O logging off
PathName = file{PROP:Profile} !Queries the name of the log file
file{PROP:Log}=string !Writes the string to the log file
file{PROP:Details}=1 !Turns Record Buffer logging on
fFile{PROP:Details}=0 !Turns Record Buffer logging off
```

where *Pathname* is the full pathname or the filename of the log file to create. If you do not specify a path, the driver writes the log file to the current directory.

You can also accomplish on demand logging with a SEND() command and the LOGFILE driver string. See *LOGFILE* for more information.

## Language Level Error Checking

When the `ERRORCODE()` procedure returns 90, you can use the `FILEERROR()` and `FILEERRORCODE()` functions to capture messages and codes returned from the backend server or file system. See the *Language Reference* for more information on these functions.

## ERROR Messages

---

All of TopSpeed's database drivers post error conditions and messages that can be accessed with the `ERRORCODE()`, `ERROR()`, `FILEERRORCODE()` and `FILEERROR()` procedures (see the *Language Reference*). The drivers post the codes immediately after each I/O (`OPEN`, `NEXT`, `GET`, `ADD`, `DELETE`, etc.) operation.

**Tip:** The `ERRORFILE()` procedure returns the file or table that produced the error.

See *Run Time Errors* in the *Language Reference* for information on the `ERRORCODE()` values and their corresponding `ERROR()` message text. The `ERROR()` text is configurable using the environment file (`CLAMSG`) and the `LOCALE` procedure. See *Internationalization*, `CLAMSG`, and `LOCALE` in the *Language Reference* for more information.

In addition, the `Btrieve`, `xBase` (`Clipper`, `dBaseIII`, `dBaseIV`, and `FoxPro`), and `ODBC FILEERROR()` message text is also configurable using the environment file (`CLAMSG`) and the `LOCALE` procedure. See the *Configurable Error Messages* section for each driver for a list of the `FILEERRORCODE` values and their corresponding `FILEERROR()` text.

## Disk Caching and Data Integrity

---

Disk caching can interfere with the data integrity features of many file systems. By disk caching, we mean any facility (for example `SMARTDRV`) that tells the database driver that a record was written to the disk when in fact it was not.

To improve performance, disk caching facilities typically accumulate several records at a time in RAM then write them to disk all at once. While this does improve performance, it can result in corrupt data files if the system fails (due to a power outage, etc.) before the records are written to disk.

A reliable Uninterruptable Power Supply (UPS) can drastically reduce this risk. Therefore, we generally recommend no disk caching, but if you must cache, then be sure to use a reliable UPS.

## Common Driver Strings

There are switches or “driver strings” you can set to control the way your application creates, reads, and writes files with a specific driver. Driver strings are simply messages or parameters that are sent to the file driver at run-time to control its behavior. The various driver specific driver strings are described in the *Driver Strings* section for each driver.

Driver strings are sent in three ways: with the OPEN or CREATE statement, with the SEND procedure, and with property syntax.

### DRIVER(‘Driver’, ‘/DriverString = value’)

The OPEN(file) and CREATE(file) statements send any driver strings specified in the FILE’s DRIVER attribute. OPEN sends the string immediately *before* the file is opened. You may specify these driver strings with a hand coded FILE declaration (see *DRIVER* in the *Language Reference* for more information) or in the Data Dictionary (**Driver Options** field in the **File Properties** dialog—see *The Dictionary Editor—File Properties* in the *User’s Guide*). In either case, you must prepend a forward slash (/) to the driver string. For example:

```
MyFile FILE,DRIVER('TopSpeed','/LOGFILE=MyFile.Log')
CODE
OPEN(MyFile) !sends the LOGFILE driver string
```

### [ MyVar = ] SEND(file, ‘DriverString [ = value ]’)

The SEND procedure sends or queries a driver string at any time, including before the file is opened. SEND may both set a new value and return the old value, or it may only set a value or only return a value. With SEND, the ISAM drivers do not require the preceding forward slash, but the SQL drivers do require it. For example:

```
SEND(MyFile,'LOGFILE='&MyLogFile) !Set logfile for ISAM
MyLogFile=SEND(MySQLFile,'/LOGFILE') !get log filename for SQL
OldLogFile=SEND(MyFile,'LOGFILE='&NewLogFile) !Set & get the logfile
```

### file{PROP:DriverString}

Property syntax is an alternative to the SEND function. With property syntax you can send a driver string to the file driver any time *after* the file is opened. With property syntax, the driver string does not require the preceding forward slash. For example:

```
MyLogFile = 'MyFile.Log'
MyFile{PROP:Profile}=MyLogFile !Set the logfile
MyLogFile = MyFile{PROP:Profile} !get the logfile filename
```

All TopSpeed database drivers support the following Driver Strings:

## ALLOWDETAILS

---

**DRIVER('Driver', '/ALLOWDETAILS = TRUE | FALSE' )**

The ALLOWDETAILS driver string allows the driver to include record buffer contents in the log file for encrypted files.

The ALLOWDETAILS driver string works with the Details switch described in the *Debugging and Tracing File I/O* section.

## Common Driver Properties

You can set or query driver properties to control the driver's behavior or to discover information about the driver. The various driver specific properties are described in the *Driver Properties* section for each driver.

You access driver properties with property syntax, for example:

```
MyFile{PROP:Profile}=MyLogFile !Set the logfile
MyLogFile = MyFile{PROP:Profile} !get the logfile filename
```

See also *VIEW and FILE Properties* in the *Language Reference*. All TopSpeed database drivers support the following Driver Properties.

### PROP:SQLDriver

---

PROP:SQLDriver returns a value indicating whether the driver understands SQL (accepts SQL statements with PROP:SQL). A value of one (1 or True) indicates the driver understands SQL; a value of zero (0 or False) indicates the driver does not understand SQL. For example:

```
SQLDriver# = Customer{PROP:SQLDriver} !query SQL capability
IF SQLDriver#
 SQLString"=Customer{PROP:SQL} !Get last SQL statement
END
```



# 24 - ASCII DATABASE DRIVER

## Specifications

The ASCII driver reads and writes standard ASCII files without field delimiters. This is often used for mainframe data import/export with an ASCII flat-file. By default, a carriage-return/line-feed delimits records. The ASCII driver does not support keys.

## Files

---

|                    |                                       |
|--------------------|---------------------------------------|
| <b>C5ASCL.LIB</b>  | Windows Static Link Library (16-bit)  |
| <b>C5ASCXL.LIB</b> | Windows Static Link Library (32-bit)  |
| <b>C5ASC.LIB</b>   | Windows Export Library (16-bit)       |
| <b>C5ASCX.LIB</b>  | Windows Export Library (32-bit)       |
| <b>C5ASC.DLL</b>   | Windows Dynamic Link Library (16-bit) |
| <b>C5ASCX.DLL</b>  | Windows Dynamic Link Library (32-bit) |

**Tip:** Due to its lack of relational features and security (anyone can view and change an ASCII file using Notepad), it's unlikely you'll use the ASCII driver to store large data files. But it can help you create a text file viewer—use it to open a file, and read it in to a multi-line edit or list box control!

## Supported Data Types

---

STRING  
GROUP

## File Specifications/Maximums

---

|                        |                            |
|------------------------|----------------------------|
| File Size:             | 4,294,967,295 bytes        |
| Records per File:      | 4,294,967,295 bytes        |
| Record Size:           | 65,520 bytes               |
| Field Size:            | 65,520 bytes               |
| Fields per Record:     | 65,520                     |
| Keys/Indexes per File: | n/a                        |
| Key Size:              | n/a                        |
| Memo fields per File:  | n/a                        |
| Memo Field Size:       | n/a                        |
| Open Data Files:       | Operating system dependent |

## Driver Strings

There are switches or “driver strings” you can set to control the way your application creates, reads, and writes files with a specific driver. Driver strings are simply messages or parameters that are sent to the file driver at run-time to control its behavior. See *Common Driver Features—Driver Strings* for an overview of these runtime Database Driver switches and parameters.

**Note:** Some driver strings have no effect after the file is open, so no SEND function syntax is listed for those strings. However, the SEND function syntax to return the value of the switch is listed for all driver strings.

The ASCII Driver supports the following Driver Strings:

### CLIP

```
DRIVER('ASCII', '/CLIP = on | off')
[Clip" =] SEND(file, 'CLIP [= on | off]')
```

The driver automatically removes trailing spaces from a record before writing it to file. Conversely, the driver automatically expands the clipped records with spaces when read. To disable this feature, set CLIP to OFF. The default is ON. SEND returns the CLIP setting (ON or OFF) in the form of a STRING(3).

### CTRLZISEOF

```
DRIVER('ASCII', '/CTRLZISEOF = on | off')
[EOF" =] SEND(file, 'CTRLZISEOF [= on | off]')
```

By default (CTRLZISEOF=on) the file driver assumes that any Ctrl-Z characters in the file indicate the end of file. To disable this feature set CTRLZISEOF=off. SEND returns the CTRLZISEOF setting in a STRING(3).

### ENDOFRECORD

```
DRIVER('ASCII', '/ENDOFRECORD = n [,m]')
[EOR" =] SEND(file, 'ENDOFRECORD [= n [,m]]')
```

Specifies the end of record delimiter.

*n* represents the number of characters that make up the end-of-record delimiter.

$m$  represents the ASCII code(s) for the end-of-record delimiter, separated by commas. The default is 2,13,10, indicating 2 characters mark the end-of-record, namely, carriage return (13) and line feed (10). SEND returns the end of record delimiter.

**Tip:** Mainframes frequently use just a carriage return to delimit records. You can use ENDOFRECORD to read these files.

## FILEBUFFERS

**DRIVER('ASCII', '/FILEBUFFERS = n')**  
**[ Buffers" = ] SEND(file, 'FILEBUFFERS [ = n ]')**

Sets the size of the buffer used to read and write to the file, where the buffer size is  $(n * 512)$  bytes. Use the /FILEBUFFERS driver string to increase the buffer size if access is slow. Maximum buffer size is 65,504 in 16-bit and 4,294,967,264 in 32-bit. SEND returns the size of the buffer in bytes.

**Tip:** The default buffer size for files opened denying write access to other users is the larger of 1024 or  $(2 * \text{record size})$ , and the larger of 512 or record size for all other open modes.

## TAB

**DRIVER('ASCII', '/TAB = n')**  
**[ Spaces" = ] SEND(file, 'TAB [ = n ]')**

Sets or queries TAB/SPACE expansion. The ASCII driver expands TABs (ASCII character 9) to spaces when reading. The value indicates the number of spaces with which to replace the tab, subject to the guidelines below. The default value is 8. SEND returns the number of spaces which replace the tab character.

*If  $n > 0$* , spaces replace each tab until the character pointer moves to the next multiple of  $n$ . For example, with the default of 8, if the TAB character is the third character in the record, 6 spaces replace the TAB.

*If  $n = 0$* , the driver removes tabs without replacement.

*If  $n < 0$* , the driver removes tabs with the positive value of  $n$  spaces. For example, "TAB=-4" causes 4 spaces to replace every tab, regardless of the position of the tab in the record.

*If  $n = -100$* , tabs remain as tabs; the driver *does not* replace them with spaces.

## QUICKSCAN

---

```
DRIVER('ASCII', '/QUICKSCAN = on | off')
[QScan" =] SEND(file, 'QUICKSCAN [= on | off]')
```

Specifies buffered access behavior. The ASCII driver reads a buffer at a time (not a record), allowing faster access. In a multi-user environment these buffers are not 100% trustworthy for subsequent access, because another user may change the file between accesses. As a safeguard, the driver rereads the buffers before each record access. To *disable* the reread, set QUICKSCAN to ON. The default is ON for files opened denying write access to other users, and OFF for all other open modes. SEND returns the Quickscan setting (ON or OFF) in the form of a STRING(3).

## Supported Attributes and Procedures

|                                                                 |                  |                                                                 |                  |
|-----------------------------------------------------------------|------------------|-----------------------------------------------------------------|------------------|
| <b>File Attributes</b>                                          | <b>Supported</b> |                                                                 |                  |
| CREATE .....                                                    | Y                | PACK( <i>file</i> ) .....                                       | N                |
| DRIVER( <i>filetype</i> [, <i>driver string</i> ]) .....        | Y                | POINTER( <i>file</i> ) .....                                    | Y <sup>2</sup>   |
| NAME .....                                                      | Y                | POINTER( <i>key</i> ) .....                                     | N                |
| ENCRYPT .....                                                   | N                | POSITION( <i>file</i> ) .....                                   | Y <sup>3</sup>   |
| OWNER( <i>password</i> ) .....                                  | N                | POSITION( <i>key</i> ) .....                                    | N                |
| RECLAIM .....                                                   | N                | RECORDS( <i>file</i> ) .....                                    | N                |
| PRE( <i>prefix</i> ) .....                                      | Y                | RECORDS( <i>key</i> ) .....                                     | N                |
| BINDABLE .....                                                  | Y                | REMOVE( <i>file</i> ) .....                                     | Y                |
| THREAD .....                                                    | Y <sup>4</sup>   | RENAME( <i>file</i> , <i>new file</i> ) .....                   | Y                |
| EXTERNAL( <i>member</i> ) .....                                 | Y                | SEND( <i>file</i> , <i>message</i> ) .....                      | Y                |
| DLL( <i>[flag]</i> ) .....                                      | Y                | SHARE( <i>file</i> , <i>access mode</i> ) .....                 | Y                |
| OEM .....                                                       | Y                | STATUS( <i>file</i> ) .....                                     | Y                |
|                                                                 |                  | STREAM( <i>file</i> ) .....                                     | N                |
|                                                                 |                  | UNLOCK( <i>file</i> ) .....                                     | Y                |
| <b>File Structures</b>                                          | <b>Supported</b> | <b>Record Access</b>                                            | <b>Supported</b> |
| INDEX .....                                                     | N                | ADD( <i>file</i> ) .....                                        | Y                |
| KEY .....                                                       | N                | ADD( <i>file</i> , <i>length</i> ) .....                        | N                |
| MEMO .....                                                      | N                | APPEND( <i>file</i> ) .....                                     | Y                |
| BLOB .....                                                      | N                | APPEND( <i>file</i> , <i>length</i> ) .....                     | N                |
| RECORD .....                                                    | Y                | DELETE( <i>file</i> ) .....                                     | N                |
|                                                                 |                  | GET( <i>file</i> , <i>key</i> ) .....                           | N                |
| <b>Index, Key, Memo Attributes</b>                              | <b>Supported</b> | GET( <i>file</i> , <i>filepointer</i> ) .....                   | Y                |
| BINARY .....                                                    | N                | GET( <i>file</i> , <i>filepointer</i> , <i>length</i> ) .....   | N                |
| DUP .....                                                       | N                | GET( <i>key</i> , <i>keypointer</i> ) .....                     | N                |
| NOCASE .....                                                    | N                | HOLD( <i>file</i> ) .....                                       | N                |
| OPT .....                                                       | N                | NEXT( <i>file</i> ) .....                                       | Y                |
| PRIMARY .....                                                   | N                | NOMEMO( <i>file</i> ) .....                                     | N                |
| NAME .....                                                      | N                | PREVIOUS( <i>file</i> ) .....                                   | N                |
| Ascending Components .....                                      | N                | PUT( <i>file</i> ) .....                                        | Y <sup>1</sup>   |
| Descending Components .....                                     | N                | PUT( <i>file</i> , <i>filepointer</i> ) .....                   | Y <sup>1</sup>   |
| Mixed Components .....                                          | N                | PUT( <i>file</i> , <i>filepointer</i> , <i>length</i> ) .....   | N                |
|                                                                 |                  | RELEASE( <i>file</i> ) .....                                    | N                |
| <b>Field Attributes</b>                                         | <b>Supported</b> | REGET( <i>file</i> , <i>string</i> ) .....                      | Y                |
| DIM .....                                                       | Y                | REGET( <i>key</i> , <i>string</i> ) .....                       | N                |
| OVER .....                                                      | Y                | RESET( <i>file</i> , <i>string</i> ) .....                      | Y                |
| NAME .....                                                      | Y                | RESET( <i>key</i> , <i>string</i> ) .....                       | N                |
|                                                                 |                  | SET( <i>file</i> ) .....                                        | Y                |
| <b>File Procedures</b>                                          | <b>Supported</b> | SET( <i>file</i> , <i>key</i> ) .....                           | N                |
| BOF( <i>file</i> ) .....                                        | N                | SET( <i>file</i> , <i>filepointer</i> ) .....                   | Y                |
| BUFFER( <i>file</i> ) .....                                     | N                | SET( <i>key</i> ) .....                                         | N                |
| BUILD( <i>file</i> ) .....                                      | N                | SET( <i>key</i> , <i>key</i> ) .....                            | N                |
| BUILD( <i>key</i> ) .....                                       | N                | SET( <i>key</i> , <i>keypointer</i> ) .....                     | N                |
| BUILD( <i>index</i> ) .....                                     | N                | SET( <i>key</i> , <i>key</i> , <i>filepointer</i> ) .....       | N                |
| BUILD( <i>index</i> , <i>components</i> ) .....                 | N                | SKIP( <i>file</i> , <i>count</i> ) .....                        | N                |
| BUILD( <i>index</i> , <i>components</i> , <i>filter</i> ) ..... | N                | WATCH( <i>file</i> ) .....                                      | N                |
| BYTES( <i>file</i> ) .....                                      | Y                |                                                                 |                  |
| CLOSE( <i>file</i> ) .....                                      | Y                | <b>Transaction Processing</b>                                   | <b>Supported</b> |
| COPY( <i>file</i> , <i>new file</i> ) .....                     | Y                | LOGOUT( <i>timeout</i> , <i>file</i> , ..., <i>file</i> ) ..... | N                |
| CREATE( <i>file</i> ) .....                                     | Y                | COMMIT .....                                                    | N                |
| DUPLICATE( <i>file</i> ) .....                                  | N                | ROLLBACK .....                                                  | N                |
| DUPLICATE( <i>key</i> ) .....                                   | N                |                                                                 |                  |
| EMPTY( <i>file</i> ) .....                                      | Y                | <b>Null Data Processing</b>                                     | <b>Supported</b> |
| EOF( <i>file</i> ) .....                                        | Y                | NULL( <i>field</i> ) .....                                      | N                |
| FLUSH( <i>file</i> ) .....                                      | N                | SETNULL( <i>field</i> ) .....                                   | N                |
| LOCK( <i>file</i> ) .....                                       | Y                | SETNONNULL( <i>field</i> ) .....                                | N                |
| NAME( <i>label</i> ) .....                                      | Y                |                                                                 |                  |
| OPEN( <i>file</i> , <i>access mode</i> ) .....                  | Y                |                                                                 |                  |

## Notes

---

- 1 When using PUT() with this driver you should take care to PUT back the same number of characters that were read. If you PUT back more characters than were read, then the “extra” characters will overwrite the first part of the subsequent record. If you PUT back fewer characters than were read, then only the first part of the retrieved record is overwritten, while the last part of the retrieved record remains as it was prior to the PUT().
- 2 POINTER() returns the relative byte position within the file.
- 3 POSITION(file) returns a STRING(4).
- 4 THREADED files consume additional file handles for each thread that accesses the file.

# 25 - BASIC DATABASE DRIVER

## Specifications

The BASIC file driver reads and writes comma-delimited ASCII files. By default, quotes ( “ ” ) surround strings, commas delimit fields, and a carriage-return/line-feed delimits records. The original BASIC programming language defined this file format. The Basic driver does not support keys or backward file processing (thus Basic files are not a good choice for random access processing).

**Tip:** The Basic file format provides a good choice for a common file format for sharing data with spreadsheet programs. A common file extension used for these files is \*.CSV, which stands for “comma separated values.”

## Files

---

|                    |                                       |
|--------------------|---------------------------------------|
| <b>C5BASL.LIB</b>  | Windows Static Link Library (16-bit)  |
| <b>C5BASXL.LIB</b> | Windows Static Link Library (32-bit)  |
| <b>C5BAS.LIB</b>   | Windows Export Library (16-bit)       |
| <b>C5BASX.LIB</b>  | Windows Export Library (32-bit)       |
| <b>C5BAS.DLL</b>   | Windows Dynamic Link Library (16-bit) |
| <b>C5BASX.DLL</b>  | Windows Dynamic Link Library (32-bit) |

## Supported Data Types

---

|         |          |
|---------|----------|
| BYTE    | DECIMAL  |
| SHORT   | PDECIMAL |
| USHORT  | STRING   |
| LONG    | CSTRING  |
| ULONG   | PSTRING  |
| SREAL   | DATE     |
| REAL    | TIME     |
| BFLOAT4 | GROUP    |
| BFLOAT8 |          |

## File Specifications/Maximums

---

|                        |                            |
|------------------------|----------------------------|
| File Size:             | 4,294,967,295 bytes        |
| Records per File:      | 4,294,967,295 bytes        |
| Record Size:           | 65,520 bytes               |
| Field Size:            | 65,520 bytes               |
| Fields per Record:     | 65,520                     |
| Keys/Indexes per File: | n/a                        |
| Key Size:              | n/a                        |
| Memo fields per File:  | 0                          |
| Memo Field Size:       | n/a                        |
| Open Data Files:       | Operating system dependent |

## Driver Strings

There are switches or “driver strings” you can set to control the way your application creates, reads, and writes files with a specific driver. Driver strings are simply messages or parameters that are sent to the file driver at run-time to control its behavior. See *Common Driver Features—Driver Strings* for an overview of these runtime Database Driver switches and parameters.

**Note:** Some driver strings have no effect after the file is open, so no SEND function syntax is listed for those strings. However, the SEND function syntax to return the value of the switch is listed for all driver strings.

The Basic Driver supports the following Driver Strings:

### ALWAYSQUOTE

```
DRIVER('BASIC', '/ALWAYSQUOTE = on | off')
[QScan" =] SEND(file, 'ALWAYSQUOTE [= on | off]')
```

For compatibility with Basic format data files created by products which do *not* place string values in quotes, set ALWAYSQUOTE to OFF.

When the contents of a string field includes the comma or quote character(s), and ALWAYSQUOTE is off, the Basic driver automatically places quotes around the string when writing to file. This also applies to delimiter characters specified with FIELDDELIMITER, or COMMA. For example, with the defaults in use and ALWAYSQUOTE off, a STRING field containing the value *1313 Mockingbird Lane, Apt. 33* is automatically stored as: “1313 Mockingbird Lane, Apt. 33”

SEND returns the ALWAYSQUOTE setting (ON or OFF) in the form of a STRING(3).

### COMMA

```
DRIVER('BASIC', '/COMMA = n')
[Comma" =] SEND(file, 'COMMA [= n]')
```

Specifies a single character field separator.

*n* represents the ANSI code for the field separator character. The default is 44, which is equivalent to “/FIELDDELIMITER=1,44.”

SEND returns the ASCII code for the field separator character.

## CTRLZISEOF

---

```
DRIVER('BASIC', '/CTRLZISEOF = on | off')
[EOF" =] SEND(file, 'CTRLZISEOF [= on | off]')
```

By default (CTRLZISEOF=on) the file driver assumes that any Ctrl-Z characters in the file indicate the end of file. To disable this feature set CTRLZISEOF=off. SEND returns the CTRLZISEOF setting in a STRING(3).

## ENDOFRECORD

---

```
DRIVER('BASIC', '/ENDOFRECORD = n [,m]')
[EOR" =] SEND(file, 'ENDOFRECORD [= n [,m]]')
```

Specifies the end of record delimiter.

*n* represents the number of characters that make up the end-of-record delimiter.

*m* represents the ANSI code(s) for the end-of-record delimiter, separated by commas. The default is 2,13,10, indicating 2 characters mark the end-of-record, namely, carriage return (13) and line feed (10). SEND returns the end of record delimiter.

**Tip:** Mainframes frequently use just a carriage return to delimit records. You can use ENDOFRECORD to read these files.

## ENDOFRECORDINQUOTE

---

```
DRIVER('BASIC', '/ENDOFRECORDINQUOTE = on | off')
[EORQuote" =] SEND(file, 'ENDOFRECORDINQUOTE [= on | off]')
```

By default (ENDOFRECORDINQUOTE=ON) the file driver does not recognize an end-of-record marker that is within a quoted string. To force End-Of-Record markers to always terminate a record, set ENDOFRECORDINQUOTE=OFF. SEND returns the ENDOFRECORDINQUOTE setting (ON or OFF) in the form of a STRING(3).

## FILEDDELIMITER

---

```
DRIVER('BASIC', '/FILEDDELIMITER = n [,m]')
[Limiter" =] SEND(file, 'FILEDDELIMITER [= n [,m]]')
```

Specifies the field separator. This is in addition to any string delimiter specified by the /QUOTE driver string.

*n* represents the number of characters that make up the field separator.

*m* represents the ANSI code(s) for the field separator characters, separated by commas. The default is 1,44 which indicates the comma character.

SEND returns the field delimiter character.

**Tip:** If both FILEDDELIMITER and COMMA are specified, the last specification prevails.

## FILEBUFFERS

---

```
DRIVER('BASIC', '/FILEBUFFERS = n')
[Buffers" =] SEND(file, 'FILEBUFFERS [= n]')
```

Sets the size of the buffer used to read and write to the file, where the buffer size is (*n* \* 512 bytes). Use the /FILEBUFFERS driver string to increase the buffer size if access is slow. Maximum buffer size is 65,504 in 16-bit and 4,294,967,264 in 32-bit. SEND returns the size of the buffer in bytes.

**Tip:** The default buffer size for files opened denying write access to other users is the larger of 1024 or (2 \* record size), and the larger of 512 or record size for all other open modes.

## QUICKSCAN

---

```
DRIVER('BASIC', '/QUICKSCAN = on | off')
[QScan" =] SEND(file, 'QUICKSCAN [= on | off]')
```

Specifies buffered access behavior. The ASCII driver reads a buffer at a time (not a record), allowing faster access. In a multi-user environment these buffers are not 100% trustworthy for subsequent access, because another user may change the file between accesses. As a safeguard, the driver rereads the buffers before each record access. To *disable* the reread, set QUICKSCAN to ON. The default is ON for files opened denying write access to other users, and OFF for all other open modes. SEND returns the QUICKSCAN setting (ON or OFF) in the form of a STRING(3).

**Tip:** TAB-delimited values are a common format compatible with the Windows clipboard. Using the BASIC file driver string /COMMA=9 lets you read Windows clipboard files

## QUOTE

---

```
DRIVER('BASIC', '/QUOTE = n')
[Quote" =] SEND(file, 'QUOTE [= n]')
```

Specifies a single character string delimiter.

*n* is the ANSI code of the delimiter character. The default is 34, the ASCII value for the quotation mark.

SEND returns the ASCII code of the single character string delimiter.

## Popular File Formats

---

The following demonstrates how to use the driver strings to create two popular file formats:

- ◆ Microsoft Word for Windows Mail Merge:

```
/ALWAYSQUOTE=OFF
/FIELDDELIMITER=1,9
/ENDOFRECORD=1,13
```

- ◆ TAB delimited format:

```
/COMMA=9
```

## Supported Attributes and Procedures

### File Attributes Supported

|                                                          |                |
|----------------------------------------------------------|----------------|
| CREATE .....                                             | Y              |
| DRIVER( <i>filetype</i> [, <i>driver string</i> ]) ..... | Y              |
| NAME .....                                               | Y              |
| ENCRYPT .....                                            | N              |
| OWNER( <i>password</i> ) .....                           | N              |
| RECLAIM .....                                            | N              |
| PRE( <i>prefix</i> ) .....                               | Y              |
| BINDABLE .....                                           | Y              |
| THREAD .....                                             | Y <sup>4</sup> |
| EXTERNAL( <i>member</i> ) .....                          | Y              |
| DLL( <i>iflag</i> ) .....                                | Y              |
| OEM .....                                                | Y              |

### File Structures Supported

|              |   |
|--------------|---|
| INDEX .....  | N |
| KEY .....    | N |
| MEMO .....   | N |
| BLOB .....   | N |
| RECORD ..... | Y |

### Index, Key, Memo, Attributes Supported

|                             |   |
|-----------------------------|---|
| BINARY .....                | N |
| DUP .....                   | N |
| NOCASE .....                | N |
| OPT .....                   | N |
| PRIMARY .....               | N |
| NAME .....                  | N |
| Ascending Components .....  | N |
| Descending Components ..... | N |
| Mixed Components .....      | N |

### Field Attributes Supported

|            |   |
|------------|---|
| DIM .....  | Y |
| OVER ..... | Y |
| NAME ..... | Y |

### File Procedures Supported

|                                                 |   |
|-------------------------------------------------|---|
| BOF( <i>file</i> ) .....                        | N |
| BUFFER( <i>file</i> ) .....                     | N |
| BUILD( <i>file</i> ) .....                      | N |
| BUILD( <i>key</i> ) .....                       | N |
| BUILD( <i>index</i> ) .....                     | N |
| BUILD( <i>index, components</i> ) .....         | N |
| BUILD( <i>index, components, filter</i> ) ..... | N |
| BYTES( <i>file</i> ) .....                      | Y |
| CLOSE( <i>file</i> ) .....                      | Y |
| COPY( <i>file, new file</i> ) .....             | Y |
| CREATE( <i>file</i> ) .....                     | Y |
| DUPLICATE( <i>file</i> ) .....                  | N |
| DUPLICATE( <i>key</i> ) .....                   | N |
| EMPTY( <i>file</i> ) .....                      | Y |
| EOF( <i>file</i> ) .....                        | Y |
| FLUSH( <i>file</i> ) .....                      | N |
| LOCK( <i>file</i> ) .....                       | Y |
| NAME( <i>label</i> ) .....                      | Y |
| OPEN( <i>file, access mode</i> ) .....          | Y |

|                                         |                |
|-----------------------------------------|----------------|
| PACK( <i>file</i> ) .....               | N              |
| POINTER( <i>file</i> ) .....            | Y <sup>2</sup> |
| POINTER( <i>key</i> ) .....             | N              |
| POSITION( <i>file</i> ) .....           | Y <sup>3</sup> |
| POSITION( <i>key</i> ) .....            | N              |
| RECORDS( <i>file</i> ) .....            | N              |
| RECORDS( <i>key</i> ) .....             | N              |
| REMOVE( <i>file</i> ) .....             | Y              |
| RENAME( <i>file, new file</i> ) .....   | Y              |
| SEND( <i>file, message</i> ) .....      | Y              |
| SHARE( <i>file, access mode</i> ) ..... | Y              |
| STATUS( <i>file</i> ) .....             | Y              |
| STREAM( <i>file</i> ) .....             | N              |
| UNLOCK( <i>file</i> ) .....             | Y              |

### Record Access Supported

|                                               |                |
|-----------------------------------------------|----------------|
| ADD( <i>file</i> ) .....                      | Y              |
| ADD( <i>file, length</i> ) .....              | N              |
| APPEND( <i>file</i> ) .....                   | Y              |
| APPEND( <i>file, length</i> ) .....           | N              |
| DELETE( <i>file</i> ) .....                   | N              |
| GET( <i>file, key</i> ) .....                 | N              |
| GET( <i>file, filepointer</i> ) .....         | Y              |
| GET( <i>file, filepointer, length</i> ) ..... | N              |
| GET( <i>key, keypointer</i> ) .....           | N              |
| HOLD( <i>file</i> ) .....                     | N              |
| NEXT( <i>file</i> ) .....                     | Y              |
| NOMEMO( <i>file</i> ) .....                   | N              |
| PREVIOUS( <i>file</i> ) .....                 | N              |
| PUT( <i>file</i> ) .....                      | Y <sup>1</sup> |
| PUT( <i>file, filepointer</i> ) .....         | Y <sup>1</sup> |
| PUT( <i>file, filepointer, length</i> ) ..... | N              |
| RELEASE( <i>file</i> ) .....                  | N              |
| REGET( <i>file, string</i> ) .....            | Y              |
| REGET( <i>key, string</i> ) .....             | N              |
| RESET( <i>file, string</i> ) .....            | Y              |
| RESET( <i>key, string</i> ) .....             | N              |
| SET( <i>file</i> ) .....                      | Y              |
| SET( <i>file, key</i> ) .....                 | N              |
| SET( <i>file, filepointer</i> ) .....         | Y              |
| SET( <i>key</i> ) .....                       | N              |
| SET( <i>key, key</i> ) .....                  | N              |
| SET( <i>key, keypointer</i> ) .....           | N              |
| SET( <i>key, key, filepointer</i> ) .....     | N              |
| SKIP( <i>file, count</i> ) .....              | N              |
| WATCH( <i>file</i> ) .....                    | N              |

### Transaction Processing Supported

|                                                 |   |
|-------------------------------------------------|---|
| LOGOUT( <i>timeout, file, ..., file</i> ) ..... | N |
| COMMIT .....                                    | N |
| ROLLBACK .....                                  | N |

### Null Data Processing Supported

|                                  |   |
|----------------------------------|---|
| NULL( <i>field</i> ) .....       | N |
| SETNULL( <i>field</i> ) .....    | N |
| SETNONNULL( <i>field</i> ) ..... | N |

## Notes

---

- 1 When using PUT() with this driver you should take care to PUT back the same number of characters that were read. If you PUT back more characters than were read, then the “extra” characters will overwrite the first part of the subsequent record. If you PUT back fewer characters than were read, then only the first part of the retrieved record is overwritten, while the last part of the retrieved record remains as it was prior to the PUT().
- 2 POINTER() returns the relative byte position within the file.
- 3 POSITION(file) returns a STRING(4).
- 4 THREADED files consume additional file handles for each thread that accesses the file.



# 26 - BTRIEVE DATABASE DRIVER

## Specifications

This file driver reads and writes Btrieve files using low-level direct access. This includes the version of Btrieve which is now part of Pervasive.SQL.

Under Clarion, the Btrieve file driver is implemented by using .DLLs and an .EXE supplied by Pervasive Software (formerly Btrieve Technologies, Inc.). For an application to use a Btrieve file driver, the following files must accompany the executable:

### 16-bit

**WBTR32.EXE**

**WBTRLOCL.DLL**

**WBTRCALL.DLL**

**WBTRVRES.DLL**

### 32-bit

You must purchase a 32-bit Btrieve engine. You can call TopSpeed at 1-800-354-5444 to order the engine.

**LICENSE WARNING: A registered Clarion owner cannot redistribute the above files outside of his/her organization without a license from Pervasive Software.**

## Files

---

|                    |                                       |
|--------------------|---------------------------------------|
| <b>C5BTRL.LIB</b>  | Windows Static Link Library (16-bit)  |
| <b>C5BTRXL.LIB</b> | Windows Static Link Library (32-bit)  |
| <b>C5BTR.LIB</b>   | Windows Export Library (16-bit)       |
| <b>C5BTRX.LIB</b>  | Windows Export Library (32-bit)       |
| <b>C5BTR.DLL</b>   | Windows Dynamic Link Library (16-bit) |
| <b>C5BTRX.DLL</b>  | Windows Dynamic Link Library (32-bit) |

## Data Types

| <u>Clarion data type</u>   | <u>Btrieve data type</u>         |
|----------------------------|----------------------------------|
| BYTE                       | STRING (1 byte)                  |
| SHORT                      | INTEGER (2 bytes)                |
| LONG                       | INTEGER (4 bytes)                |
| SREAL                      | FLOAT (4 bytes)                  |
| REAL                       | FLOAT (8 bytes)                  |
| BFLOAT4                    | BFLOAT (4 bytes)                 |
| BFLOAT8                    | BFLOAT (8 bytes)                 |
| PDECIMAL                   | DECIMAL                          |
| STRING                     | STRING                           |
| CSTRING                    | ZSTRING                          |
| PSTRING                    | LSTRING                          |
| DATE                       | DATE                             |
| TIME                       | TIME                             |
| USHORT                     | UNSIGNED BINARY (2 bytes)        |
| ULONG                      | UNSIGNED BINARY (4 bytes)        |
| MEMO                       | STRING, LVAR or NOTE (see below) |
| BYTE, NAME('LOGICAL')      | LOGICAL*                         |
| USHORT, NAME('LOGICAL')    | LOGICAL*                         |
| PDECIMAL, NAME('MONEY')    | MONEY*                           |
| STRING(@N0n-), NAME('STS') | SIGNED TRAILING SEPARATE*        |
| DECIMAL*                   |                                  |

### Notes:

- \* You can store Clarion DECIMAL types in a Btrieve file. However, you cannot build a key or index using the field. This is provided for backward compatibility with older Clarion programs which used the Btrieve LEM. If you need standard Btrieve decimal data that is compatible with any other Btrieve compliant program, you should use the PDECIMAL data type and not the DECIMAL data type.
- \* If you want to create a file with LOGICAL or MONEY field types, you must specify LOGICAL or MONEY in the field's NAME attribute. If you are accessing an existing file, the NAME attribute is not required.

LOGICAL may be declared as a BYTE or USHORT, depending on whether it is a one or two byte LOGICAL:

```
LogicalField1 BYTE !One byte LOGICAL
LogicalField2 USHORT !Two byte LOGICAL
```

MONEY may be declared as a PDECIMAL(x,2), where x is the total number of digits to be stored:

```
MoneyField PDECIMAL(7,2),NAME('MONEY') !Store up to 99999.99
```

- \* Btrieve NUMERIC fields are not fully supported by the driver. Btrieve NUMERIC is stored as a string with the last character holding a digit and an implied sign. The possible values for this last character are:

```
 1 2 3 4 5 6 7 8 9 0
Positive: A B C D E F G H I {
Negative: J K L M N O P Q R }
```

To access a NUMERIC field you must define a STRING(@N0x), where

x is one less than the digits in the NUMERIC, and a STRING(1) to hold the sign indicator. The Btrieve driver does not maintain this sign field, the application must be written to directly handle it.

For example to access a NUMERIC(7) you would have:

```

NumericGroup GROUP !Store -999999 to 999999
Number STRING(@N06) !Numbers
Sign STRING(1) !Sign indicator
END

```

## File Specifications/Maximums

---

|                        |   |                                                                                                                                                                                         |
|------------------------|---|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| File Size              | : | 4,000,000,000 bytes                                                                                                                                                                     |
| Records per File       | : | Limited by the size of the file                                                                                                                                                         |
| Record Size            |   |                                                                                                                                                                                         |
| Client-based           | : | 65,535 bytes variable length                                                                                                                                                            |
| Server based           | : | 57,000 bytes variable length                                                                                                                                                            |
| Field Size             | : | 65,520 bytes                                                                                                                                                                            |
| Fields per Record      | : | 65,520                                                                                                                                                                                  |
| Keys/Indexes per File: |   | 24 with NLM5<br>256 with NLM6.<br>Client Btrieve v6.15                                                                                                                                  |
|                        |   | <u>Page Size</u> <u>Max Key Segments</u>                                                                                                                                                |
|                        |   | 512                    8                                                                                                                                                                |
|                        |   | 1,024                 23                                                                                                                                                                |
|                        |   | 1,536                 24                                                                                                                                                                |
|                        |   | 2,048                 54                                                                                                                                                                |
|                        |   | 4,096                 119                                                                                                                                                               |
|                        |   | This is the total number of components.<br>If you have a multicomponent key built<br>from three fields, this counts as three<br>indexes when counting the number of<br>allowed indexes. |
| Key Size               | : | 255 bytes                                                                                                                                                                               |
| Memo fields per File:  |   | System memory dependent                                                                                                                                                                 |
| Memo field size        | : | 65,520 bytes                                                                                                                                                                            |
| Open Files             | : | Operating system dependent                                                                                                                                                              |

**Note:** The Btrieve driver supports data only and key only files.

## Driver Strings

There are switches or “driver strings” you can set to control the way your application creates, reads, and writes files with a specific driver. Driver strings are simply messages or parameters that are sent to the file driver at run-time to control its behavior. See *Common Driver Features—Driver Strings* for an overview of these runtime Database Driver switches and parameters.

The Btrieve Driver supports the following Driver Strings:

### ACS

---

```
DRIVER('BTRIEVE', '/ACS = filename')
[SortSeq" =] SEND(file, 'ACS [= filename]')
```

When creating a Btrieve file you can specify an alternate collating sequence for sorting STRING keys. This sorting sequence is normally obtained from the sort sequence you define in the INI file for your program. However, Btrieve supplies files for doing case insensitive sorts. To create your file using these sort sequences you specify the name of the sort file in the driver string.

### ALLOWREAD

---

```
DRIVER('BTRIEVE', '/ALLOWREAD = ON | OFF')
[Read" =] SEND(file, 'ALLOWREAD [= ON | OFF]')
```

By default, a Btrieve file created with an owner name may be accessed *only* in read-only mode when the owner name is not known. To prevent *all* access to the file without the owner name, set ALLOWREAD to OFF. SEND returns the ALLOWREAD setting (ON or OFF) in the form of a STRING(3).

### APPENDBUFFER

---

```
DRIVER('BTRIEVE', '/APPENDBUFFER = size ')
[Buffer" =] SEND(file, 'APPENDBUFFER [= size]')
```

By default, APPEND adds records to the file one at a time. To get better performance over a network you can tell the driver to build up a buffer of records then send all of them to Btrieve at once. *Size* is the number of records you want to allocate for the buffer. SEND returns the number of records that will fit in the buffer.

## BALANCEKEYS

---

```
DRIVER('BTRIEVE', '/BALANCEKEYS = ON | OFF')
[Balance" =] SEND(file, 'BALANCEKEYS [= ON | OFF]')
```

When creating a Btrieve file, you can use this driver string to tell Btrieve that all keys associated with the file must be stored in a balanced btree. This saves disk space, but will slow down file adds, deletes and updates where key values change. SEND returns the BALANCEKEYS setting (ON or OFF) in the form of a STRING(3).

## COMPRESS

---

```
DRIVER('BTRIEVE', '/COMPRESS = ON | OFF')
[Read" =] SEND(file, 'COMPRESS [= ON | OFF]')
```

Btrieve lets you compress the data before storage. This allows for a smaller storage requirement, but reduces performance. When COMPRESS is ON, CREATE creates a compressed Btrieve file. SEND returns the COMPRESS setting (ON or OFF) in the form of a STRING(3).

## FREESPACE

---

```
DRIVER('BTRIEVE', '/FREESPACE = 0 | 10 | 20 | 30')
[Read" =] SEND(file, 'FREESPACE [= 0 | 10 | 20 | 30]')
```

Specifies the percentage of free space to maintain on variable length pages. The default is zero. SEND returns the percentage of free space to maintain on variable length pages.

## LACS

---

```
DRIVER('BTRIEVE', '/LACS [= | country_id,codepage]')
[Sequence" =] SEND(file, 'LACS [= | country_id,codepage]')
```

Btrieve v6.15 and later offers the Local Alternate Collating Sequences feature. This allows your string keys to sort based on the country code for the machine running your program. To use this feature you put '/LACS=' in your driver string.

### /LACS=country\_ID,code\_page

You can also specify a User-Defined Alternate Collating Sequence. This allows your string key to sort based on the DOS country code and code page for a particular country. To use this feature you put '/LACS=country\_id,codepage' in your driver string. Note that there must be no spaces surrounding the comma.

SEND returns country\_id,codepage or the string ',' if using the Local Alternate Collating Sequences feature.

## MEMO

---

```
DRIVER('BTREIVE', '/MEMO = SINGLE | LVAR | NOTE [,delimiter]')
[Memo" =] SEND(file, 'MEMO [= SINGLE | LVAR | NOTE [,delimiter]]')
```

### /MEMO=SINGLE

To access existing Btrieve files created with the Btrieve LEM from Clarion Professional Developer 2.1(DOS), or files with variable length records set MEMO to SINGLE.

### /MEMO=LVAR

To access a file with variable length records, use a SINGLE style MEMO whose size equals the maximum size of the variable length component of the record. To add/put records to this style file with binary data stored in the variable length section, use the ADD(file,length), APPEND(file,length) and PUT(file,pos,length) functions. The driver ignores the *pos* parameter in the PUT function, but initialize it to 0 (*zero*) for future compatibility. The ADD, APPEND or PUT functions will remove all trailing spaces for text memos and NULL characters for binary memos before storing the record.

### /MEMO=NOTE,<delimiter>

To access Xtrieve data files that have data type of Note or LVar, set the driver string to NOTE and LVAR respectively. With the NOTE data type, specify the end-of-field delimiter. Specify the ASCII value for the delimiter. NOTE and LVAR memos do not require the use of the size variants of ADD, APPEND and PUT, when storing records. The end of record marker is not necessary for a NOTE style memo. The driver automatically adds the end of record marker before storing the record and removes it before putting the memo data into the memo buffer.

As an example, “/MEMO=NOTE,141” indicates a file with an Xtrieve Notes field using CR/LF as the delimiter. For more information on the Xtrieve data types refer to the documentation supplied by Novell.

### SEND(file,'MEMO')

Returns the MEMO setting: NORMAL, NOTE, LVAR, or SINGLE.

## PAGESIZE

---

```
DRIVER('BTRIEVE', '/PAGESIZE=SIZE')
[PSize" =] SEND(file, 'PAGESIZE[=SIZE')
```

Sets the Btrieve Page size at file creation time. The size must be a multiple of 512, with a maximum of 4096. Larger page sizes usually result in more efficient disk storage. *Do not add spaces before or after the equal sign.*

SEND returns the page size setting.

## PREALLOCATE

---

```
DRIVER('BTRIEVE', '/PREALLOCATE = n')
[Read" =] SEND(file, 'PREALLOCATE [= n]')
```

When creating a Btrieve file, you can preallocate *n* pages of disk space for the file. The default is zero. SEND returns the number of pages of allocated disk space.

## TRUNCATE

---

```
DRIVER('BTRIEVE', '/TRUNCATE = ON | OFF')
[Trunc" =] SEND(file, 'TRUNCATE [= ON | OFF]')
```

When creating a Btrieve file, you can use this driver string to tell Btrieve to truncate trailing spaces. This forces the record to be stored as a variable length record. SEND returns the TRUNCATE setting (ON or OFF) in the form of a STRING(3).

## Driver Properties

You can use Clarion's property syntax to query and set certain SQL Accelerator driver properties. These properties are described below.

### PROP:PositionBlock

---

PROP:PositionBlock returns the Btrieve pointer to the Btrieve position block used by the Btrieve driver for the named file. This allows you to call Btrieve operations directly. For example:

```
MAP
 MODULE('Btrieve')
 BTRV(USHORT, LONG, <*STRING>, *UNSIGNED, <*STRING>, BYTE, BYTE), |
 NAME('BTRV'), PASCAL, RAW
 END
END

StatData STRING(33455)
KeyData STRING(64)
DataLen UNSIGNED(33455)

CODE
PosBlock = file{PROP:PositionBlock}
BTRV(15, PosBlock, StatData, DataLen, KeyData, 64, 0) !Get file statistics
```

## Supported Attributes and Procedures

### File Attributes Supported

|                                                          |                 |
|----------------------------------------------------------|-----------------|
| CREATE .....                                             | Y               |
| DRIVER( <i>filetype</i> [, <i>driver string</i> ]) ..... | Y               |
| NAME .....                                               | Y               |
| ENCRYPT .....                                            | Y               |
| OWNER( <i>password</i> ) .....                           | Y <sup>1</sup>  |
| RECLAIM .....                                            | Y               |
| PRE( <i>prefix</i> ) .....                               | Y               |
| BINDABLE .....                                           | Y               |
| THREAD .....                                             | Y <sup>15</sup> |
| EXTERNAL( <i>member</i> ) .....                          | Y               |
| DLL( <i>flag</i> ) .....                                 | Y               |
| OEM .....                                                | Y               |

### File Structures Supported

|              |                |
|--------------|----------------|
| INDEX .....  | Y              |
| KEY .....    | Y              |
| MEMO .....   | Y <sup>2</sup> |
| BLOB .....   | N              |
| RECORD ..... | Y              |

### Index, Key, Memo Attributes Supported

|                             |                 |
|-----------------------------|-----------------|
| BINARY .....                | Y <sup>16</sup> |
| DUP .....                   | Y               |
| NOCASE .....                | Y               |
| OPT .....                   | Y               |
| PRIMARY .....               | Y               |
| NAME .....                  | Y <sup>2</sup>  |
| Ascending Components .....  | Y               |
| Descending Components ..... | Y               |
| Mixed Components .....      | Y               |

### Field Attributes Supported

|            |   |
|------------|---|
| DIM .....  | Y |
| OVER ..... | Y |
| NAME ..... | Y |

### File Procedures Supported

|                                                 |                 |
|-------------------------------------------------|-----------------|
| BOF( <i>file</i> ) .....                        | Y <sup>10</sup> |
| BUFFER( <i>file</i> ) .....                     | N               |
| BUILD( <i>file</i> ) .....                      | Y <sup>3</sup>  |
| BUILD( <i>key</i> ) .....                       | Y <sup>3</sup>  |
| BUILD( <i>index</i> ) .....                     | Y <sup>3</sup>  |
| BUILD( <i>index, components</i> ) .....         | Y <sup>3</sup>  |
| BUILD( <i>index, components, filter</i> ) ..... | N               |
| BYTES( <i>file</i> ) .....                      | N               |
| CLOSE( <i>file</i> ) .....                      | Y               |
| COPY( <i>file, new file</i> ) .....             | Y               |
| CREATE( <i>file</i> ) .....                     | Y               |
| DUPLICATE( <i>file</i> ) .....                  | Y               |
| DUPLICATE( <i>key</i> ) .....                   | Y               |
| EMPTY( <i>file</i> ) .....                      | Y               |
| EOF( <i>file</i> ) .....                        | Y <sup>10</sup> |
| FLUSH( <i>file</i> ) .....                      | Y               |
| LOCK( <i>file</i> ) .....                       | N <sup>4</sup>  |
| NAME( <i>label</i> ) .....                      | Y               |
| OPEN( <i>file, access mode</i> ) .....          | Y               |

|                                         |                 |
|-----------------------------------------|-----------------|
| PACK( <i>file</i> ) .....               | Y               |
| POINTER( <i>file</i> ) .....            | Y <sup>11</sup> |
| POINTER( <i>key</i> ) .....             | Y <sup>11</sup> |
| POSITION( <i>file</i> ) .....           | Y <sup>12</sup> |
| POSITION( <i>key</i> ) .....            | Y <sup>12</sup> |
| RECORDS( <i>file</i> ) .....            | Y               |
| RECORDS( <i>key</i> ) .....             | Y               |
| REMOVE( <i>file</i> ) .....             | Y               |
| RENAME( <i>file, new file</i> ) .....   | Y               |
| SEND( <i>file, message</i> ) .....      | Y               |
| SHARE( <i>file, access mode</i> ) ..... | Y               |
| STATUS( <i>file</i> ) .....             | Y               |
| STREAM( <i>file</i> ) .....             | Y               |
| UNLOCK( <i>file</i> ) .....             | N               |

### Record Access Supported

|                                               |                  |
|-----------------------------------------------|------------------|
| ADD( <i>file</i> ) .....                      | Y <sup>5</sup>   |
| ADD( <i>file, length</i> ) .....              | Y <sup>5</sup>   |
| APPEND( <i>file</i> ) .....                   | Y <sup>6</sup>   |
| APPEND( <i>file, length</i> ) .....           | Y <sup>5,6</sup> |
| DELETE( <i>file</i> ) .....                   | Y <sup>7</sup>   |
| GET( <i>file, key</i> ) .....                 | Y                |
| GET( <i>file, filepointer</i> ) .....         | Y                |
| GET( <i>file, filepointer, length</i> ) ..... | N                |
| GET( <i>key, keypointer</i> ) .....           | Y                |
| HOLD( <i>file</i> ) .....                     | Y                |
| NEXT( <i>file</i> ) .....                     | Y                |
| NOMEMO( <i>file</i> ) .....                   | Y                |
| PREVIOUS( <i>file</i> ) .....                 | Y                |
| PUT( <i>file</i> ) .....                      | Y <sup>5</sup>   |
| PUT( <i>file, filepointer</i> ) .....         | N                |
| PUT( <i>file, filepointer, length</i> ) ..... | Y                |
| RELEASE( <i>file</i> ) .....                  | Y                |
| REGET( <i>file, string</i> ) .....            | Y                |
| REGET( <i>key, string</i> ) .....             | Y                |
| RESET( <i>file, string</i> ) .....            | Y                |
| RESET( <i>key, string</i> ) .....             | Y                |
| SET( <i>file</i> ) .....                      | Y                |
| SET( <i>file, key</i> ) .....                 | Y                |
| SET( <i>file, filepointer</i> ) .....         | Y <sup>8</sup>   |
| SET( <i>key</i> ) .....                       | Y                |
| SET( <i>key, key</i> ) .....                  | Y                |
| SET( <i>key, keypointer</i> ) .....           | Y <sup>8</sup>   |
| SET( <i>key, key, filepointer</i> ) .....     | Y <sup>9</sup>   |
| SKIP( <i>file, count</i> ) .....              | Y                |
| WATCH( <i>file</i> ) .....                    | Y                |

### Transaction Processing Supported

|                                                 |                    |
|-------------------------------------------------|--------------------|
| LOGOUT( <i>timeout, file, ..., file</i> ) ..... | Y <sup>13,14</sup> |
| COMMIT .....                                    | Y <sup>14</sup>    |
| ROLLBACK .....                                  | Y                  |

### Null Data Processing Supported

|                                  |   |
|----------------------------------|---|
| NULL( <i>field</i> ) .....       | N |
| SETNULL( <i>field</i> ) .....    | N |
| SETNONNULL( <i>field</i> ) ..... | N |

## Notes

---

- 1 We recommend using a variable password that is lengthy and contains special characters because this more effectively hides the password value from anyone looking for it. For example, a password like “dd...#\$....\*&” is much more difficult to “find” than a password like “SALARY.”

**Tip:** To specify a variable instead of the actual password in the Owner Name field of the File Properties dialog, type an exclamation point (!) followed by the variable name. For example: !MyPassword.

- 2 The driver ignores any NAME attribute on a MEMO field. MEMO fields can reside either in a separate file, or in the data file if the driver string /MEMO is set to SINGLE, LVAR or NOTE. If the driver string /MEMO is not set, the separate MEMO file name is “MEM,” preceded by the first five characters of the file’s label, plus the file extension “.DAT.” Setting the driver string /MEMO restricts you to one memo field per file.
- 3 If used after an APPEND(), but before a file is closed, this adds the keys dropped by APPEND(). In all other cases BUILD() rebuilds the file and keys. If you only want to rebuild keys, doing a BUILD(key) for each key is faster than BUILD(file).
- 4 Btrieve does not directly support file locking. If you require file locking, use LOGOUT.
- 5 When using the LVAR and NOTE memo type, make certain that the memo has the appropriate structure. If the structure is incorrect and the driver calculates a length greater than the maximum memo size defined for that file, these functions fail and set errorcode to 57 - Invalid Memo File.
- 6 Btrieve does not support non-key updates. To emulate APPEND() behavior, the driver drops all indexes possible when APPEND() is first called. Calling BUILD() immediately after appending records rebuilds the dropped key fields.
- 7 Btrieve’s DELETE destroys positioning information when processing in file order. The driver attempts to reposition to the appropriate record. This is not always possible and may require the driver to read from the start of the file. Using key order processing avoids this possible slow down.
- 8 If a file pointer or key pointer has a value of zero, the driver ignores the pointer parameter. Processing is set to either file or key order, and the record pointer is set to the first element.
- 9 If the file pointer has a value of zero, processing starts at the first key value whose position is greater than (or less than for PREVIOUS) the file pointer. Not passing a valid pointer, other than maximum LONG or maximum ULONG, is inefficient.

- 10 These functions are supported, but not recommended. They cause more disk I/O than `ERRORCODE()`. `Btrieve` returns *eof* when reading past the last record. Therefore, the driver must read the next record, then the *next* to see if it's at the end of file, then return to the record you want.
- 11 `POINTER()` returns a relative position within the file, not a record number.
- 12 `POSITION(file)` returns a `STRING(4)`. `POSITION(key)` returns a `STRING` the size of the key fields + 4 bytes.
- 13 If a system crashes during a transaction (`LOGOUT—COMMIT`), the recovery is automatically handled by the `Btrieve` driver the next time the affected file is accessed.

When you issue a `LOGOUT`, all `Btrieve` files accessed during the transaction are logged out. The following code is illegal because you cannot close a logged-out file:

```
LOGOUT(1, file1)
OPEN(file1)
CLOSE(file1)
```

- 14 See also *PROP:Logout* in the *Language Reference*.
- 15 `THREADED` files do not consume additional file handles for each thread that accesses the file.
- 16 OEM conversion is not applied to `BINARY MEMOs`. The driver assumes `BINARY MEMOs` are zero padded; otherwise, space padded.

## Other

### Client/Server

For Client/Server-based Btrieve, Netware Btrieve is a server-based version of Btrieve that runs on a Novell server.

### File Structure

A single file normally holds the data and all keys. Data filenames default to a \*.DAT file extension. By default, the driver stores memos in a separate file, or optionally in the data file itself, given the appropriate driver string.

Because Btrieve is a data-model independent, indexed record manager, it does not store field definitions within the data itself. The application accessing the data determines how to interpret the Btrieve record based on Btrieve Data Definition Files (.DDF). Absent .DDF files describing the Btrieve file, it is very difficult for an application meaningfully interpret Btrieve data.

The Btrieve file format stores minimal file structure information in the file. As far as possible, the driver validates your description against the information in the file. However, it *is* possible to successfully open a Btrieve file that has key definitions that do not exactly match your definition. You must make certain that your file and key definitions accurately match the Btrieve file.

### Keys and Indexes

KEYs are dynamic, and automatically update when the data file changes.

INDEXes are stored separately from data files. INDEX files receive a temporary file name, and are deleted when the program terminates normally. INDEXes are static—they are not automatically updated when the data file changes. The BUILD statement creates or updates index files.

### Record Lengths

The driver stores records less than 4K as fixed length. It stores records greater than 4K as variable length. The minimum record length is 4 bytes. One record can be held in each open file by each user.

### Page Size

To determine the physical record length, add 8 bytes for each KEY that allows duplicates. Add 4 bytes if the file allows variable record lengths. Finally, allow 6 bytes for overhead per page.

For example: If the record size is 300 bytes and the file has three KEYS that allow Duplicates, the total record size is:

$$\begin{array}{rcl}
 & 300 & \text{record size} \\
 \times & 24 & \text{overhead for three KEYS with the DUP attribute} \\
 = & 324 & \text{physical record length}
 \end{array}$$

A page size of 512 would only hold one such record, and 182 bytes per page would go unused (512 - 6 - 324). If the page size were 1024, three records could be stored per page and only 46 bytes would go unused (1024 - 6 - (324 \* 3)).

You must load BTRIEVE.EXE with a page size equal to or greater than the largest page size of any file that you will be accessing.

### File Sharing

Btrieve lets you open a file in five different formats: NORMAL, ACCELERATED, READ-ONLY, VERIFY, or EXCLUSIVE. The equivalent Clarion OPEN() states are:

| <u>Btrieve State</u> | <u>Clarion OPEN/SHARE access mode</u>                                                                     |
|----------------------|-----------------------------------------------------------------------------------------------------------|
| ACCELERATED          | Read/Write with FCB compatibility mode (2H)                                                               |
| READ-ONLY            | Read Only (0H,10H,20H,30H,40H)                                                                            |
| VERIFY               | Write Only with FCB compatibility mode (1H)                                                               |
| EXCLUSIVE            | Write Only with any Deny flag (11H,21H,31H,41H);<br>Read/Write with Deny All, Read or Write (12H,22H,32H) |
| NORMAL               | Read/Write with Deny None (42H)                                                                           |

Btrieve allows a file to have a specified owner. See the /READONLY driver string for details on setting this flag. The file may also be encrypted with the ENCRYPT attribute. A file can only be encrypted when an owner name is supplied.

### Record Pointers

Btrieve uses an unsigned long for its internal record pointer; negative values are stripped of their sign. We recommend the ULONG data type for your record pointer.

### Collating Sequence

- ❖ Key Attribute: NOCASE

NLM 5 does not support case insensitive indexing. When necessary, you must supply an alternate collating sequence which implements case insensitive sorting.

Btrieve supports an alternate collating sequence. However, NLM 6 does not support *both* NOCASE *and* an alternate collating sequence. If you

specify both, the NOCASE attribute takes precedence. No error is returned from the SEND function.

- ❖ **Changing the Collating Sequence**

Btrieve stores the collating sequence inside the file. So to change the collating sequence you have to change the .ENV file, then create a new Btrieve file based on the modified .ENV file, then copy the data from the old file to the new file.

### **KEY Definitions**

- ❖ When defining a file, the key definition does not need to exactly match the underlying file. For example, you can have a physical file with a single component STRING(20). You can define this as a key with two string components with a total length of 20. The rule is that the data types must match and the total size must match. However, if your Clarion definition does not exactly match the underlying file, the driver cannot optimize APPEND() or BUILD() statements.
- ❖ A Key's NAME attribute can add additional functionality.

#### **KEY,NAME('MODIFIABLE=true|false')**

Btrieve lets you create a key that cannot be changed once created. To use this feature you can use the name attribute on the key to set MODIFIABLE to FALSE. It defaults to TRUE.

#### **KEY,NAME('ANYNULL')**

Btrieve lets you create a key that will not include a record if any key components are null. To create such a key you specify ANYNULL in the key name.

For example, to create a key that is non-modifiable and excludes keys if any component is null:

```
Key1 KEY(+pre:field1,-pre:field2),NAME('ANYNULL MODIFIABLE=FALSE')
```

#### **KEY,NAME('AUTOINCREMENT')**

The Clarion CREATE statement creates a Btrieve autoincrement key.

#### **KEY,NAME('REPEATINGDUPLICATE')**

By default Btrieve version 6 stores a reference to only the first record in a series of duplicate records in a key. The other occurrences of the duplicate key value are obtained by following a link list stored at the record. To create an index where all duplicate records are stored in the key you use the NAME('REPEATINGDUPLICATE'). This produces larger keys, but random access to duplicate records is faster (this feature is only available for version 6 files).

## Configurable **ERROR** Messages

All of TopSpeed's database drivers post error conditions and messages that can be accessed with the `ERRORCODE()`, `ERROR()`, `FILEERRORCODE()` and `FILEERROR()` procedures (see the *Language Reference*). The drivers post the codes immediately after each I/O (`OPEN`, `NEXT`, `GET`, `ADD`, `DELETE`, etc.) operation.

**Tip:** The `ERRORFILE()` procedure returns the file or table that produced the error.

See *Run Time Errors* in the *Language Reference* for information on the `ERRORCODE()` values and their corresponding `ERROR()` message text. The `ERROR()` text is configurable using the environment file (`CLAMSG`) and the `LOCALE` procedure. See *Internationalization*, `CLAMSG`, and `LOCALE` in the *Language Reference* for more information.

In addition, Btrieve driver `FILEERROR()` message text is also configurable using the environment file (`CLAMSG`) and the `LOCALE` procedure. However, the `CLAMSG` value required to change the message text is equal to the `FILEERRORCODE()` value + 1000. Following is a list of the `FILEERROR()` text, the `FILEERRORCODE()` values, and the `CLAMSG` values required to change the `FILEERROR()` text:

| <code>FILEERRORCODE</code> | <code>CLAMSG</code> | <code>FILEERROR</code>          |
|----------------------------|---------------------|---------------------------------|
| 1                          | 1001                | Invalid Operation               |
| 2                          | 1002                | IO Error                        |
| 3                          | 1003                | File Not Open                   |
| 4                          | 1004                | Key value Not Found             |
| 5                          | 1005                | Duplicate Key Value             |
| 6                          | 1006                | Invalid Key Number              |
| 7                          | 1007                | Different Key Number            |
| 8                          | 1008                | Position Not Set                |
| 9                          | 1009                | End of File                     |
| 10                         | 1010                | Modifiable Key Value Error      |
| 11                         | 1011                | Bad File Name                   |
| 12                         | 1012                | File Not Found                  |
| 13                         | 1013                | Extended File Error             |
| 14                         | 1014                | Pre-Image Open Error            |
| 15                         | 1015                | Pre-Image I/O Error             |
| 16                         | 1016                | Expansion Error                 |
| 17                         | 1017                | Close Error                     |
| 18                         | 1018                | Disk Full                       |
| 19                         | 1019                | Unrecoverable Error             |
| 20                         | 1020                | Btrieve Record Manager Inactive |
| 21                         | 1021                | Key Buffer Too Short            |
| 22                         | 1022                | Data Buffer Length              |
| 23                         | 1023                | Position Block Length           |

## FILEERRORCODE CLAMSG

## FILEERROR

|    |      |                                       |
|----|------|---------------------------------------|
| 24 | 1024 | Page Size Error                       |
| 25 | 1025 | Create IO Error                       |
| 26 | 1026 | Number Of Keys                        |
| 27 | 1027 | Invalid Key Position                  |
| 28 | 1028 | Invalid Record Length                 |
| 29 | 1029 | Invalid Key Length                    |
| 30 | 1030 | Not A Btrieve File                    |
| 31 | 1031 | File Already Extended                 |
| 32 | 1032 | Extend I/O Error                      |
| 33 | 1033 | Unknown Error Code                    |
| 34 | 1034 | Invalid Extension Name                |
| 35 | 1035 | Directory Error                       |
| 36 | 1036 | Transaction Error                     |
| 37 | 1037 | Transaction Active                    |
| 38 | 1038 | Transaction Control File I/O Error    |
| 39 | 1039 | End/Abort Transaction Error           |
| 40 | 1040 | Transaction Max Files                 |
| 41 | 1041 | Operation Not Allowed                 |
| 42 | 1042 | Incomplete Accelerated Access         |
| 43 | 1043 | Invalid Record Address                |
| 44 | 1044 | Null Key Path                         |
| 45 | 1045 | Inconsistent Key Flags                |
| 46 | 1046 | Access to File Denied                 |
| 47 | 1047 | Maximum Open Files                    |
| 48 | 1048 | Invalid Alternate Sequence Definition |
| 49 | 1049 | Key Type Error                        |
| 50 | 1050 | Owner Already Set                     |
| 51 | 1051 | Invalid Owner                         |
| 52 | 1052 | Error Writing Cache                   |
| 53 | 1053 | Invalid Interface                     |
| 54 | 1054 | Variable Page Error                   |
| 55 | 1055 | Autoincrement Error                   |
| 56 | 1056 | Incomplete Index                      |
| 57 | 1057 | Expanded Memory Error                 |
| 58 | 1058 | Compression Buffer Too Short          |
| 59 | 1059 | File Already Exists                   |
| 60 | 1060 | Reject Count Reached                  |
| 61 | 1061 | Small Ex Get Buffer Error             |
| 62 | 1062 | Invalid Get Expression                |
| 63 | 1063 | Invalid Ext Insert Buffer             |
| 64 | 1064 | Optimise Limit Reached                |
| 65 | 1065 | Invalid Extractor                     |
| 66 | 1066 | RI Too Many Databases                 |
| 67 | 1067 | RI DDF Cannot Open                    |
| 68 | 1068 | RI Cascade Too Deep                   |
| 69 | 1069 | RI Cascade Error                      |
| 71 | 1071 | RI Violation                          |
| 72 | 1072 | RI Reference File Cannot Open         |

**FILEERRORCODE CLAMSG****FILEERROR**

|      |      |                                   |
|------|------|-----------------------------------|
| 73   | 1073 | RI Out of Sync                    |
| 74   | 1074 | End Changed to Abort              |
| 76   | 1076 | RI Conflict                       |
| 77   | 1077 | Cannot Loop in Server             |
| 78   | 1078 | Dead Lock                         |
| 79   | 1079 | Programming Error                 |
| 80   | 1080 | Conflict                          |
| 81   | 1081 | Lock Error                        |
| 82   | 1082 | Lost Position                     |
| 83   | 1083 | Read Outside Transaction          |
| 84   | 1084 | Record in Use                     |
| 85   | 1085 | File in Use                       |
| 86   | 1086 | File Table Full                   |
| 87   | 1087 | Handle Table Full                 |
| 88   | 1088 | Incompatible Mode Error           |
| 90   | 1090 | Redirected Device Table Full      |
| 91   | 1091 | Server Error                      |
| 92   | 1092 | Transaction Table Full            |
| 93   | 1093 | Incompatible Lock Type            |
| 94   | 1094 | Permission Error                  |
| 95   | 1095 | Session No Longer Valid           |
| 96   | 1096 | Communications Environment        |
| 97   | 1097 | Data Message Too Small            |
| 98   | 1098 | Internal Transaction Error        |
| 100  | 1100 | No Cache Memory Available         |
| 101  | 1101 | No OS Memeory Available           |
| 102  | 1102 | No Stack Available                |
| 103  | 1103 | Chunk Offset Too Long             |
| 104  | 1104 | Locale Error                      |
| 105  | 1105 | Cannot Create with Bat            |
| 106  | 1106 | Chunk cannot Get Next             |
| 107  | 1107 | Chunk incompatible File           |
| 1001 | 2001 | Lock Parameter Out of Range       |
| 1002 | 2002 | Memory Allocation Error           |
| 1003 | 2003 | Mem Parameter Too Small           |
| 1004 | 2004 | Page Size Parameter Out of Range  |
| 1005 | 2005 | Invalid Preimage Parameter        |
| 1006 | 2006 | Preimage Buff Param Out of Range  |
| 1007 | 2007 | Files Parameter Out of Range      |
| 1008 | 2008 | Invalid Init Parameter            |
| 1009 | 2009 | Invalid Trans Parameter           |
| 1010 | 2010 | Error Accessing Tran Control File |
| 1011 | 2011 | Compression Buf Parm Out of Range |
| 1013 | 2013 | Task List Full                    |
| 1014 | 2014 | Stop Warning                      |
| 1016 | 2016 | Already Iniialised                |
| 2001 | 3001 | Insufficient Memory Allocated     |
| 2002 | 3002 | Invalid Option                    |

**FILEERRORCODE CLAMSG**

2003  
2004  
2005  
2006  
2007

3003  
3004  
3005  
3006  
3007

**FILEERROR**

No Local Access Allowed  
SPX Not Installed  
Incorrect SPX Version  
No Available SPX Connection  
Invalid Pointer Parameter

# 27 - CLARION DATABASE DRIVER

## Specifications

The Clarion file driver is compatible with the file system used by Clarion for DOS 3.1 and Clarion Professional Developer 2.1, patch 2.109 and later.

Keys and Indexes exist as separate files from the data file. Keys are dynamic—they are automatically updated as the data file changes. The default file extension for a key file is \*.K##. Indexes are static—they do not automatically update, but instead require the BUILD statement for updating.

The driver stores records as fixed length. It stores memo fields in a separate file. The memo file defaults to the first eight characters of the File Label plus an extension of .MEM.

## Files

---

|                    |                                       |
|--------------------|---------------------------------------|
| <b>C5CLAL.LIB</b>  | Windows Static Link Library (16-bit)  |
| <b>C5CLAXL.LIB</b> | Windows Static Link Library (32-bit)  |
| <b>C5CLA.LIB</b>   | Windows Export Library (16-bit)       |
| <b>C5CLAX.LIB</b>  | Windows Export Library (32-bit)       |
| <b>C5CLA.DLL</b>   | Windows Dynamic Link Library (16-bit) |
| <b>C5CLAX.DLL</b>  | Windows Dynamic Link Library (32-bit) |

**Tip:** By avoiding the ASCII-only file formats of many other popular PC database application development systems, the Clarion file format provides a more secure means of storing data.

## Data Types

---

|       |                           |
|-------|---------------------------|
| BYTE  | DECIMAL <sup>1</sup>      |
| SHORT | STRING (255 byte maximum) |
| LONG  | MEMO                      |
| REAL  | GROUP                     |

1 Decimal sizes greater than 15 are not supported by Clarion Professional Developer 2.1.

## Maximum File Specifications

---

|                        |                            |
|------------------------|----------------------------|
| File Size:             | limited only by disk space |
| Records per File :     | 4,294,967,295              |
| Record Size:           | 65,520 bytes               |
| Field Size :           | 65,520 bytes               |
| Field Name:            | 12 characters              |
| Fields per Record:     | 65,520                     |
| Keys/Indexes per File: | 251                        |
| Key Size :             | 245 bytes                  |
| Memo fields per File : | 1                          |
| Memo Field Size:       | 65,520 bytes               |
| Open Data Files:       | Operating system dependent |

## Driver Strings

There are switches or “driver strings” you can set to control the way your application creates, reads, and writes files with a specific driver. Driver strings are simply messages or parameters that are sent to the file driver at run-time to control its behavior. See *Common Driver Features—Driver Strings* for an overview of these runtime Database Driver switches.

**Note:** Some driver strings have no effect after the file is open, so no SEND function syntax is listed for those strings. However, the SEND function syntax to return the value of the switch is listed for all driver strings.

The Clarion Driver supports the following Driver Strings:

### DELETED

---

#### SEND(file, 'DELETED')

For use only with the SEND command when IGNORESTATUS is on. Reports the status of the loaded record. If deleted, the return string is “ON” and if not “OFF.”

### HELD

---

#### SEND(file, 'HELD')

For use only with the SEND command when IGNORESTATUS is on. Reports the status of the loaded record. If held, the return string is “ON” and if not “OFF.”

### IGNORECORRUPTIONS

---

#### DRIVER('Clarion', '/IGNORECORRUPTIONS = on | off' ) [ Status" = ] SEND(file, 'IGNORECORRUPTIONS [ = on | off ]')

The driver detects key file header corruptions where the number of deleted records in the file header is different than in the key file header. By default (IGNORECORRUPTIONS=OFF), the driver posts ERRORCODE()=46 when it OPENS a file with such a corruption. To have the driver ignore these (sometimes benign) corruptions, set IGNORECORRUPTIONS=ON.

SEND returns the IGNORECORRUPTIONS setting (ON or OFF) in the form of a STRING(3).

## IGNORESTATUS

---

```
DRIVER('Clarion', '/IGNORESTATUS = on | off')
[Status" =] SEND(file, 'IGNORESTATUS [= on | off]')
```

When set *on*, the driver does *not* skip deleted records when accessing the file with GET(), NEXT(), and PREVIOUS() in file order. It also enables a PUT() on a deleted or held record. IGNORESTATUS requires opening the file in exclusive mode. However, any MEMO data of the deleted records is not recoverable. SEND returns the IGNORESTATUS setting (ON or OFF) in the form of a STRING(3).

## MAINTAINHEADERTIME

---

```
DRIVER('Clarion', '/MAINTAINHEADERTIME = on | off')
[Status" =] SEND(file, 'MAINTAINHEADERTIME [= on | off]')
```

When set *on*, the driver maintains the file header time stamp (last updated) under all circumstances. When set to *off* (the default), the driver improves performance by ignoring the time stamp under some circumstances. SEND returns the MAINTAINHEADERTIME setting (ON or OFF) in the form of a STRING(3).

## RECOVER

---

```
SEND(file, 'RECOVER = n')
```

The RECOVER string, when *n* is greater than 0, UNLOCKS data files, RELEASES held records, and rolls back incomplete transactions in order to recover from a system crash. See also *Transaction Processing for Clarion Files*.

*n* represents the number of seconds to wait before invoking the recovery process. When *n* is equal to 1, the recovery process is invoked immediately. When *n* is equal to 0, the recovery process is disarmed.

There are two ways of using RECOVER:

```
SEND(file, RECOVER=n)
OPEN(file)
```

This releases a LOCK on a file that was locked when a machine crashed. It also rolls back a transaction that was in process when a system crashed.

```
SEND(file, RECOVER=n)
GET or NEXT or PREVIOUS
```

This removes a hold flag from records that were held when a machine crashed. Here is a piece of code that removes all hold flags from a file:

```
OPEN(file) !make sure no one else is using the file
SEND(file,'IGNORESTATUS=ON')
SET(file)
LOOP
 NEXT(file)
 IF ERRORCODE() THEN BREAK.
 IF SEND(file,'HELD') = 'ON' THEN
 SEND(file,'RECOVER=1')
 REGET(file,POSITION(file))
 . .
```

**RECOVER** may not be used as a **DRIVER** string—you may only use it with the **SEND** function. The **SEND** function returns a blank string.

## Supported Attributes and Procedures

### File Attributes Supported

|                                                    |                |
|----------------------------------------------------|----------------|
| CREATE                                             | Y              |
| DRIVER( <i>filetype</i> [, <i>driver string</i> ]) | Y              |
| NAME                                               | Y              |
| ENCRYPT                                            | Y              |
| OWNER( <i>password</i> )                           | Y <sup>1</sup> |
| RECLAIM                                            | Y              |
| PRE( <i>prefix</i> )                               | Y              |
| BINDABLE                                           | Y              |
| THREAD                                             | Y <sup>8</sup> |
| EXTERNAL( <i>member</i> )                          | Y              |
| DLL( <i>[flag]</i> )                               | Y              |
| OEM                                                | Y              |

### File Structures Supported

|        |   |
|--------|---|
| INDEX  | Y |
| KEY    | Y |
| MEMO   | Y |
| BLOB   | N |
| RECORD | Y |

### Index, Key, Memo Attributes Supported

|                       |                |
|-----------------------|----------------|
| BINARY                | Y <sup>9</sup> |
| DUP                   | Y              |
| NOCASE                | Y              |
| OPT                   | Y              |
| PRIMARY               | Y              |
| NAME                  | Y              |
| Ascending Components  | Y              |
| Descending Components | N              |
| Mixed Components      | N              |

### Field Attributes Supported

|      |   |
|------|---|
| DIM  | Y |
| OVER | Y |
| NAME | Y |

### File Procedures Supported

|                                           |                |
|-------------------------------------------|----------------|
| BOF( <i>file</i> )                        | Y <sup>2</sup> |
| BUFFER( <i>file</i> )                     | N              |
| BUILD( <i>file</i> )                      | Y              |
| BUILD( <i>key</i> )                       | Y              |
| BUILD( <i>index</i> )                     | Y              |
| BUILD( <i>index, components</i> )         | Y              |
| BUILD( <i>index, components, filter</i> ) | N              |
| BYTES( <i>file</i> )                      | Y              |
| CLOSE( <i>file</i> )                      | Y              |
| COPY( <i>file, new file</i> )             | Y              |
| CREATE( <i>file</i> )                     | Y              |
| DUPLICATE( <i>file</i> )                  | Y              |
| DUPLICATE( <i>key</i> )                   | Y              |
| EMPTY( <i>file</i> )                      | Y              |
| EOF( <i>file</i> )                        | Y <sup>2</sup> |
| FLUSH( <i>file</i> )                      | Y              |
| LOCK( <i>file</i> )                       | Y              |
| NAME( <i>label</i> )                      | Y              |
| OPEN( <i>file, access mode</i> )          | Y              |

|                                   |                |
|-----------------------------------|----------------|
| PACK( <i>file</i> )               | Y              |
| POINTER( <i>file</i> )            | Y <sup>3</sup> |
| POINTER( <i>key</i> )             | Y <sup>3</sup> |
| POSITION( <i>file</i> )           | Y <sup>4</sup> |
| POSITION( <i>key</i> )            | Y <sup>4</sup> |
| RECORDS( <i>file</i> )            | Y              |
| RECORDS( <i>key</i> )             | Y              |
| REMOVE( <i>file</i> )             | Y              |
| RENAME( <i>file, new file</i> )   | Y              |
| SEND( <i>file, message</i> )      | Y              |
| SHARE( <i>file, access mode</i> ) | Y              |
| STATUS( <i>file</i> )             | Y              |
| STREAM( <i>file</i> )             | Y              |
| UNLOCK( <i>file</i> )             | Y              |

### Record Access Supported

|                                         |                 |
|-----------------------------------------|-----------------|
| ADD( <i>file</i> )                      | Y <sup>10</sup> |
| ADD( <i>file, length</i> )              | N               |
| APPEND( <i>file</i> )                   | Y               |
| APPEND( <i>file, length</i> )           | N               |
| DELETE( <i>file</i> )                   | Y               |
| GET( <i>file, key</i> )                 | Y               |
| GET( <i>file, filepointer</i> )         | Y               |
| GET( <i>file, filepointer, length</i> ) | N               |
| GET( <i>key, keypointer</i> )           | Y               |
| HOLD( <i>file</i> )                     | Y               |
| NEXT( <i>file</i> )                     | Y               |
| NOMEMO( <i>file</i> )                   | Y               |
| PREVIOUS( <i>file</i> )                 | Y               |
| PUT( <i>file</i> )                      | Y <sup>10</sup> |
| PUT( <i>file, filepointer</i> )         | Y               |
| PUT( <i>file, filepointer, length</i> ) | N               |
| RELEASE( <i>file</i> )                  | Y               |
| REGET( <i>file, string</i> )            | Y               |
| REGET( <i>key, string</i> )             | Y               |
| RESET( <i>file, string</i> )            | Y               |
| RESET( <i>key, string</i> )             | Y               |
| SET( <i>file</i> )                      | Y               |
| SET( <i>file, key</i> )                 | Y               |
| SET( <i>file, filepointer</i> )         | Y               |
| SET( <i>key</i> )                       | Y               |
| SET( <i>key, key</i> )                  | Y               |
| SET( <i>key, keypointer</i> )           | Y               |
| SET( <i>key, key, filepointer</i> )     | Y               |
| SKIP( <i>file, count</i> )              | Y               |
| WATCH( <i>file</i> )                    | Y               |

### Transaction Processing Supported<sup>5</sup>

|                                           |                  |
|-------------------------------------------|------------------|
| LOGOUT( <i>timeout, file, ..., file</i> ) | Y <sup>6,7</sup> |
| COMMIT                                    | Y                |
| ROLLBACK                                  | Y                |

### Null Data Processing Supported

|                            |   |
|----------------------------|---|
| NULL( <i>field</i> )       | N |
| SETNULL( <i>field</i> )    | N |
| SETNONNULL( <i>field</i> ) | N |

## Notes

---

- 1 We recommend using a variable password that is lengthy and contains special characters because this more effectively hides the password value from anyone looking for it. For example, a password like “dd...#\$...\*&” is much more difficult to “find” than a password like “SALARY.”

**Tip:** To specify a variable instead of the actual password in the Owner Name field of the File Properties dialog, type an exclamation point (!) followed by the variable name. For example: !MyPassword.

- 2 These functions are supported but not recommended due to the lack of support in other file systems. NEXT and PREVIOUS post Error 33 if you attempt to read beyond the end of the file.
- 3 POINTER() returns a record number.
- 4 POSITION(file) returns a STRING(4). POSITION(key) returns a STRING the size of the key fields + 4 bytes.
- 5 The RECOVER switch must be “armed” at the beginning of your program in order to support transaction processing. See *Driver Strings* for more information on the RECOVER function.
- 6 LOGOUT has the effect of LOCKing the file. See also *PROP:Logout* in the *Language Reference*.
- 7 You cannot LOGOUT an aliased file and its primary file at the same time.
- 8 THREADED files consume additional file handles for each thread that accesses the file.
- 9 OEM conversion is not applied to BINARY MEMOs. The driver assumes BINARY MEMOs are zero padded; otherwise, space padded.
- 10 Prior to Clarion 2.003 in 16-bit programs under Microsoft operating systems, writes (ADD, PUT) did not correctly flush operating system buffers. This problem is corrected with Clarion 2.003 and higher, so that writes are slower but safer. To implement the pre 2.003 behavior, use STREAM and FLUSH.

## Other

### Transaction Processing for Clarion Files

---

When you issue a LOGOUT statement the Clarion file driver creates a transaction file called programname.TR\$. By default this file is created in the same directory as the program. To create the transaction file elsewhere, add a CWC21 section to the WIN.INI file as follows:

```
[CWC21]
CLATMP=path
```

where path is a directory visible to all users. This statement

```
PUTINI('CWC21','CLATMP',path)
```

creates the correct .INI file section.

During a transaction datafile.LOG files are created for each data file edited during the transaction. These LOG files always reside in the same directory as the data file.

If a system crashes while a transaction is active, no user will be able to access the files until a recovery is run on the files. See the RECOVER send command on how to do this.

LOGOUT has the effect of LOCKING the file.

### Field Labels

---

The Clarion driver only supports fully qualified field names (prefix + label) of 16 characters or less. That is, within the Clarion file (\*.DAT) header, the driver truncates prefix + label to the first 16 characters. If the first 16 characters are not unique, the truncation results in duplicate field names.

Duplicate field names within the file header can cause problems with Clarion for DOS 2.1 and earlier. In addition, it can cause problems if you import the file definition from the Clarion file (\*.DAT), then compile a Clarion application based on the imported file definition containing the duplicate field names.

You can avoid duplicate field name problems by using the NAME attribute (the **External Name** field in the Data Dictionary's **Field Properties** dialog) to provide unique names for fields whose first 16 characters are duplicated. By providing unique names in the NAME attribute, your application can refer to the field by its (long) label, and the Clarion driver uses the unique NAME attribute to resolve conflicts.

# 28 - CLIPPER DATABASE DRIVER

## Specifications

The Clipper file driver is compatible with Clipper Summer '87 and Clipper 5.0. The default data file extension is \*.DBF.

Keys and Indexes exist as separate files from the data file. Keys are dynamic—they automatically update as the data file changes. Indexes are static—they do not automatically update, but instead require the BUILD statement for updating. The default file extension for the index file is \*.NTX.

The driver stores records as fixed length. It stores memo fields in a separate file. The memo file name takes the first eight characters of the File Label plus an extension of .DBT.

## Files

---

|                    |                                       |
|--------------------|---------------------------------------|
| <b>C5CLPL.LIB</b>  | Windows Static Link Library (16-bit)  |
| <b>C5CLPXL.LIB</b> | Windows Static Link Library (32-bit)  |
| <b>C5CLPLIB</b>    | Windows Export Library (16-bit)       |
| <b>C5CLPXLIB</b>   | Windows Export Library (32-bit)       |
| <b>C5CLP.DLL</b>   | Windows Dynamic Link Library (16-bit) |
| <b>C5CLPX.DLL</b>  | Windows Dynamic Link Library (32-bit) |

**Tip:** As a popular xBase database application development system, Clipper provides a common file format for many installed business applications and their data files. Use the Clipper driver to access these files in their native format.

## Data Types

---

The xBase file format stores all data as ASCII strings. You may either specify STRING types with declared pictures for each field, or specify native Clarion data types, which the driver converts automatically.

| <u>Clipper data type</u> | <u>Clarion data type</u> | <u>STRING w/ picture</u> |
|--------------------------|--------------------------|--------------------------|
| Date                     | DATE                     | STRING(@D12)             |
| *Numeric                 | REAL                     | STRING(@N-_p.d)          |
| *Logical                 | BYTE                     | STRING(1)                |
| Character                | STRING                   | STRING                   |
| *Memo                    | MEMO                     | MEMO                     |

If your application reads and writes to existing files, a pictured STRING will suffice. However, if your application *creates* a Clipper file, you may require additional information for these Clipper types:

- ❖ To create a numeric field in the Data Dictionary, choose the REAL data type. In the External Name field on the Attributes tab, specify '*NumericFieldName*=N(*Precision*,*DecimalPlaces*)' where *NumericFieldName* is the name of the field, *Precision* is the precision of the field and *DecimalPlaces* is the number of decimal places. With a REAL data type, you cannot access the Character or Places fields in the Field definition, you must specify those attributes with an expression in the External Name Field on the Attributes tab.

For example, if you want to create a field called Number with nine significant digits and two decimal places, enter 'Number=N(9,2)' in the External Name field on the Attributes tab of the Field properties in the Data Dictionary.

If you're hand coding a native Clarion data type, add the NAME attribute using the same syntax.

If you're hand coding a STRING with picture, STRING(@N-\_9.2), NAME('Number'), where *Number* is the field name.

- ❖ To create a logical field, using the data dictionary, choose the BYTE data type. There are no special steps; however, see the miscellaneous section for tips on reading the data from the field.

If you're hand coding a STRING with picture, add the NAME attribute: STRING(1), NAME('LogFld = L').

- ❖ To create a date field, using the data dictionary, choose the DATE data type, rather than LONG, which you usually use for the TopSpeed or Clarion file formats.
- ❖ MEMO field declarations require the a pointer field in the file's record structure. Declare the pointer field as a STRING(10) or a LONG. This field will be stored in the .DBF file containing the offset of the memo in the .DBT file. The MEMO declaration must have a NAME() attribute naming the pointer field. An example file declaration follows:

```
File FILE, DRIVER('Clipper')
Memo1 MEMO(200),NAME('Notes')
Memo2 MEMO(200),NAME('Text')
Rec RECORD
Mem1Ptr LONG,NAME('Notes')
Mem2Ptr STRING(10),NAME('Text')
END
END
```

**Tip:** Whenever possible, use the File Import Utility in the Dictionary Editor to define your files.

## File Specifications/Maximums

---

|                        |                                                        |
|------------------------|--------------------------------------------------------|
| File Size:             | 2,000,000,000 bytes                                    |
| Records per File:      | 1,000,000,000                                          |
| Record Size:           | 4,000 bytes (Clipper '87)<br>8,192 bytes (Clipper 5.0) |
| Field Size             |                                                        |
| Character:             | 254 bytes (Clipper '87)<br>2048 bytes (Clipper 5.0)    |
| Date:                  | 8 bytes                                                |
| Logical:               | 1 byte                                                 |
| Numeric:               | 20 bytes including decimal point                       |
| Memo:                  | 65,520 bytes (see note)                                |
| Fields per Record:     | 1024                                                   |
| Keys/Indexes per File: | No Limit                                               |
| Key Sizes              |                                                        |
| Character:             | 100 bytes                                              |
| Numeric, Date:         | 8 bytes                                                |
| Memo fields per File:  | Dependent on available memory                          |
| Open Files:            | Operating system dependent                             |

## Driver Strings

There are switches or “driver strings” you can set to control the way your application creates, reads, and writes files with a specific driver. Driver strings are simply messages or parameters that are sent to the file driver at run-time to control its behavior. See *Common Driver Features—Driver Strings* for an overview of these runtime Database Driver switches and parameters.

**Note:** Some driver strings have no effect after the file is open, so no SEND function syntax is listed for those strings. However, the SEND function syntax to return the value of the switch is listed for all driver strings.

The Clipper Driver supports the following Driver Strings:

## BUFFERS

---

```
DRIVER('CLIPPER', '/BUFFERS = n')
[Status" =] SEND(file, 'BUFFERS [= n]')
```

Sets the size of the buffer used to read and write to the file, where the buffer size is (n \* 512 bytes). Use the /BUFFERS driver string to increase the buffer size if access is slow. Maximum buffer size is 65,504 in 16-bit and 4,294,967,264 in 32-bit. SEND returns the size of the buffer in bytes.

**Tip:** The default is three buffers of 1024 bytes each. Increasing the number of buffers will not increase performance when a file is shared by multiple users.

## RECOVER

---

```
DRIVER('CLIPPER', '/RECOVER')
[Status" =] SEND(file, 'RECOVER')
```

Equivalent to the Xbase RECALL command, which recovers records marked for deletion. When using the Clipper driver, the DELETE statement flags a record as “inactive.” The driver does not remove the record until the PACK command is executed.

RECOVER is evaluated each time you open the file if you add the driver string to the data dictionary. When the driver recovers the records previously marked for deletion, you must manually rebuild keys and indexes with the BUILD statement.

## IGNORESTATUS

---

```
DRIVER('CLIPPER', '/IGNORESTATUS = on | off ')
[Status" =] SEND(file, 'IGNORESTATUS [on | off]')
```

When set *on*, the driver does *not* skip deleted records when accessing the file with GET, NEXT, and PREVIOUS in file order. It also enables a PUT on a deleted or held record. IGNORESTATUS requires opening the file in exclusive mode. SEND returns the IGNORESTATUS setting (ON or OFF) in the form of a STRING(3).

## DELETED

---

```
[Status" =] SEND(file, 'DELETED')
```

For use only with the SEND command, when IGNORESTATUS is on. Returns the status of the current record. If deleted, the return string is "ON" and if not, "OFF."

## Supported Attributes and Procedures

### File Attributes Supported

|                                                          |                 |
|----------------------------------------------------------|-----------------|
| CREATE .....                                             | Y <sup>1</sup>  |
| DRIVER( <i>filetype</i> [, <i>driver string</i> ]) ..... | Y               |
| NAME .....                                               | Y               |
| ENCRYPT .....                                            | N               |
| OWNER( <i>password</i> ) .....                           | N               |
| RECLAIM .....                                            | N <sup>2</sup>  |
| PRE( <i>prefix</i> ) .....                               | Y               |
| BINDABLE .....                                           | Y               |
| THREAD .....                                             | Y <sup>16</sup> |
| EXTERNAL( <i>member</i> ) .....                          | Y               |
| DLL( <i>[flag]</i> ) .....                               | Y               |
| OEM .....                                                | Y               |

### File Structures Supported

|              |                |
|--------------|----------------|
| INDEX .....  | Y              |
| KEY .....    | Y              |
| MEMO .....   | Y <sup>3</sup> |
| BLOB .....   | N              |
| RECORD ..... | Y              |

### Index, Key, Memo Attributes Supported

|                             |                |
|-----------------------------|----------------|
| BINARY .....                | N              |
| DUP .....                   | Y <sup>4</sup> |
| NOCASE .....                | Y              |
| OPT .....                   | N              |
| PRIMARY .....               | Y              |
| NAME .....                  | Y <sup>3</sup> |
| Ascending Components .....  | Y              |
| Descending Components ..... | Y              |
| Mixed Components .....      | Y              |

### Field Attributes Supported

|            |                |
|------------|----------------|
| DIM .....  | N              |
| OVER ..... | Y              |
| NAME ..... | Y <sup>1</sup> |

### File Procedures Supported

|                                                 |                 |
|-------------------------------------------------|-----------------|
| BOF( <i>file</i> ) .....                        | Y <sup>11</sup> |
| BUFFER( <i>file</i> ) .....                     | N               |
| BUILD( <i>file</i> ) .....                      | Y               |
| BUILD( <i>key</i> ) .....                       | Y               |
| BUILD( <i>index</i> ) .....                     | Y               |
| BUILD( <i>index, components</i> ) .....         | Y <sup>5</sup>  |
| BUILD( <i>index, components, filter</i> ) ..... | N               |
| BYTES( <i>file</i> ) .....                      | N               |
| CLOSE( <i>file</i> ) .....                      | Y               |
| COPY( <i>file, new file</i> ) .....             | Y <sup>6</sup>  |
| CREATE( <i>file</i> ) .....                     | Y <sup>1</sup>  |
| DUPLICATE( <i>file</i> ) .....                  | Y               |
| DUPLICATE( <i>key</i> ) .....                   | Y               |
| EMPTY( <i>file</i> ) .....                      | Y               |
| EOF( <i>file</i> ) .....                        | Y <sup>11</sup> |
| FLUSH( <i>file</i> ) .....                      | Y               |
| LOCK( <i>file</i> ) .....                       | Y               |
| NAME( <i>label</i> ) .....                      | Y               |
| OPEN( <i>file, access mode</i> ) .....          | Y <sup>7</sup>  |

|                                         |                 |
|-----------------------------------------|-----------------|
| PACK( <i>file</i> ) .....               | Y               |
| POINTER( <i>file</i> ) .....            | Y <sup>12</sup> |
| POINTER( <i>key</i> ) .....             | Y <sup>12</sup> |
| POSITION( <i>file</i> ) .....           | Y <sup>13</sup> |
| POSITION( <i>key</i> ) .....            | Y <sup>13</sup> |
| RECORDS( <i>file</i> ) .....            | Y <sup>14</sup> |
| RECORDS( <i>key</i> ) .....             | Y <sup>14</sup> |
| REMOVE( <i>file</i> ) .....             | Y               |
| RENAME( <i>file, new file</i> ) .....   | Y <sup>8</sup>  |
| SEND( <i>file, message</i> ) .....      | Y               |
| SHARE( <i>file, access mode</i> ) ..... | Y <sup>7</sup>  |
| STATUS( <i>file</i> ) .....             | Y               |
| STREAM( <i>file</i> ) .....             | Y               |
| UNLOCK( <i>file</i> ) .....             | Y               |

### Record Access Supported

|                                               |                 |
|-----------------------------------------------|-----------------|
| ADD( <i>file</i> ) .....                      | Y <sup>9</sup>  |
| ADD( <i>file, length</i> ) .....              | N               |
| APPEND( <i>file</i> ) .....                   | Y <sup>9</sup>  |
| APPEND( <i>file, length</i> ) .....           | N               |
| DELETE( <i>file</i> ) .....                   | Y <sup>6</sup>  |
| GET( <i>file, key</i> ) .....                 | Y               |
| GET( <i>file, filepointer</i> ) .....         | Y               |
| GET( <i>file, filepointer, length</i> ) ..... | N               |
| GET( <i>key, keypointer</i> ) .....           | Y               |
| HOLD( <i>file</i> ) .....                     | Y <sup>10</sup> |
| NEXT( <i>file</i> ) .....                     | Y               |
| NOMEMO( <i>file</i> ) .....                   | Y               |
| PREVIOUS( <i>file</i> ) .....                 | Y               |
| PUT( <i>file</i> ) .....                      | Y               |
| PUT( <i>file, filepointer</i> ) .....         | Y               |
| PUT( <i>file, filepointer, length</i> ) ..... | N               |
| RELEASE( <i>file</i> ) .....                  | Y               |
| REGET( <i>file, string</i> ) .....            | Y               |
| REGET( <i>key, string</i> ) .....             | Y               |
| RESET( <i>file, string</i> ) .....            | Y               |
| RESET( <i>key, string</i> ) .....             | Y               |
| SET( <i>file</i> ) .....                      | Y               |
| SET( <i>file, key</i> ) .....                 | Y               |
| SET( <i>file, filepointer</i> ) .....         | Y               |
| SET( <i>key</i> ) .....                       | Y               |
| SET( <i>key, key</i> ) .....                  | Y               |
| SET( <i>key, keypointer</i> ) .....           | Y               |
| SET( <i>key, key, filepointer</i> ) .....     | Y               |
| SKIP( <i>file, count</i> ) .....              | Y               |
| WATCH( <i>file</i> ) .....                    | Y               |

### Transaction Processing Supported<sup>15</sup>

|                                                 |   |
|-------------------------------------------------|---|
| LOGOUT( <i>timeout, file, ..., file</i> ) ..... | N |
| COMMIT .....                                    | N |
| ROLLBACK .....                                  | N |

### Null Data Processing Supported

|                                  |   |
|----------------------------------|---|
| NULL( <i>field</i> ) .....       | N |
| SETNULL( <i>field</i> ) .....    | N |
| SETNONNULL( <i>field</i> ) ..... | N |

## Notes

---

- 1 If your application *creates* a Clipper file, you may require additional NAME information for these Clipper data types:

For a Clipper numeric field, use the Clarion REAL data type. Then in the NAME attribute (the **External Name** field on the **Attributes** tab in the **Field Properties** dialog), specify `'NumericFieldName=N(Precision,DecimalPlaces)'` where *NumericFieldName* is the name of the field, *Precision* is the precision of the field and *DecimalPlaces* is the number of decimal places. See *Data Types* above for more information.

For a Clipper logical field, use the Clarion BYTE data type. See *Data Types* above for more information. See the *Miscellaneous* section for tips on reading the data from the field.

For a Clipper date field, use the Clarion DATE data type. See *Data Types* above for more information.

- 2 When the driver deletes a record from a Clipper database, the record is not physically removed, instead the driver marks it inactive. Memo fields are not physically removed from the memo file, however they cannot be retrieved if they refer to an inactive record. To remove records and memo fields permanently, execute a PACK(file).

**Tip:** To those programmers familiar with Clipper, this driver processes deleted records consistent with the way Clipper processes them after the SET DELETED ON command is issued. Records marked for deletion are ignored from processing by executable code statements, but remain in the data file.

- 3 MEMO field declarations require a pointer field in the file's record structure. Declare the pointer field as a STRING(10) or a LONG. This field will be stored in the .DBF file containing the offset of the memo in the .DBT file. The MEMO declaration must have a NAME() attribute naming the pointer field. See *Data Types* above for more information.
- 4 In Clipper it is legal to enter multiple records with duplicates of the unique key components. However, only the first of these records is indexed. So processing in key order only shows this first record. If you delete a record, then enter a new record with the same key value, the key file continues to point at the deleted record rather than the new record. In this situation, the Clipper file driver changes the key file to point at the active record rather than the deleted record. This means that if you use a Clipper program to delete a unique record, then insert a duplicate of this record, the new record is invisible when processing in key order until a pack is done. If you do the same process in a Clarion program, the new record is visible when processing in key order.

- 5 When building dynamic indexes, the *components* may take one of two forms:

```
BUILD(DynNdx, '+Pre:FLD1, -Pre:FLD2')
```

This form specifies the names of the fields on which to build the index. The field names must appear as specified in the fields' NAME() attribute if supplied, or must be the label name. A prefix may be used for compatibility with Clarion conventions but is ignored.

```
BUILD(DynNdx, 'T[Expression]')
```

This form specifies the type and Expression used to build an index—see *Miscellaneous—Key Definition* below.

- 6 The COPY() command copies data and memo files using *newfile*, which may specify a new file name or directory. Key or index files are copied if the *newfile* is a subdirectory specification. To copy an index file to a new file, use a special form of the copy command:

```
COPY(file, '<index>|<newfile>')
```

This returns *File Not Found* if an invalid index is passed. The COPY command assumes a default extension of .NTX for both the source and the target file names if none is specified. If you require a file name without an extension, terminate the name with a period. Given the file structure:

```
CLar2 FILE,CREATE,DRIVER('CLipper'),PRE(CL2)
NumKey KEY(+CL2:Num),DUP
StrKey KEY(+CL2:Str1)
StrKey2 KEY(+CL2:Str2)
AMemo MEMO(100), NAME('mem')
Record RECORD
Num STRING(@n-_9.2)
STR1 STRING(2)
STR2 STRING(2)
Mem STRING(10)
```

The following commands copy this file definition to A:

```
COPY(CLar2, 'A:\CLAR2')
COPY(CLar2, 'StrKey|A:\STRKEY')
COPY(CLar2, 'StrKey2|A:\STRKEY2')
COPY(CLar2, 'NumKey|A:\NUMKEY')
```

After these calls, the following files would exist on drive A: CLAR2.DBF, CLAR2.DBT, STRKEY.NTX, STRKEY2.NTX, and NUMKEY.NTX.

- 7 You do not need SHARE (or VSHARE) in any environment (for example, Novell) that supplies file locking as part of the operating system.
- 8 The RENAME command copies the data and memo files using *newfile*, which may specify a new file name or directory path. Key and index files must be renamed using the same syntax as the COPY command, above.

- 9 The ADD statement tests for duplicate keys before modifying the data file or its associated KEY files. Consequently it is slower than APPEND which performs no checks and does not update KEYS. When adding large amounts of data to a database use APPEND...BUILD in preference to ADD.
- 10 Clipper performs record locking by locking the entire record within the data file. This prevents read access to other processes. Therefore we recommend minimizing the amount of time for which a record is held.
- 11 Although the driver supports these functions, we do not recommend their use. They must physically access the files and add overhead. Instead, test the value returned by ERRORCODE() after each sequential access. NEXT or PREVIOUS post Error 33 (Record Not Available) if an attempt is made to access a record beyond the end or beginning of the file.
- 12 There is no distinction between file pointers and key pointers; they both return the record number for any given record.
- 13 POSITION(file) returns a STRING(12). POSITION(key) returns a STRING the size of the key fields + 4 bytes.
- 14 Under Clipper, the RECORDS() function reports the same number of records for the data file and its keys and indexes. Usually there will be no difference in the number of records *unless* the INDEX is out of date. Because the DELETE statement does not physically remove records, *the number of records reported by the RECORDS() function includes inactive records*. Exercise care when using this function.
- 15 You should clear the **Enclose RI code in transaction frame** box on the **File Control** tab of the **Global Properties** dialog when using this driver with template generated applications.
- 16 THREADED files consume additional file handles for each thread that accesses the file.

## Other

### Boolean Evaluation

- ❖ Clipper allows a logical field to accept one of nine possible values (y,Y,n,N,t,T,f,F or a space character). The space character is neither true nor false. When using a logical field from a preexisting database in a logical expression, account for all these possibilities. Remember that when a STRING field is used as an expression, it is true if it contains any data and false if it is equal to zero or blank. Therefore, to evaluate a Logical field's truth, the expression should be true if the field contains any of the "true" characters (T,t,Y, or y). For example, if a Logical field were used to specify a product as taxable or nontaxable, the expression to evaluate its truth would be:

*(If Condition):*

```
Taxable='T' OR Taxable='t' OR Taxable='Y' OR Taxable='y'
```

### Large MEMOS

- ❖ Clarion supports MEMO fields up to a maximum of 64K. If you have an existing file which includes a memo greater than 64K, you can use the file but not modify the large MEMOS.
- ❖ You can determine when your application encounters a large MEMO by detecting when the memo pointer variable is non-blank, but the memo appears to be blank. Error 47 (Bad Record Declaration) is posted. If you attempt to update such a record, any modification to the MEMO field is ignored.

### Long Field Names

- ❖ Clipper supports a maximum of 10 characters in a field name. If you require more, use an External Name with 10 characters or less.

### Sort Sequence

- ❖ The Clipper driver supports international sort orders, however, to maintain compatibility with Clipper's international sort order, remove the CLADIGRAPH= line from ..\Clarion4\BIN\Clarion4.ENV file.

### Key Definition

- ❖ Clipper supports the use of expressions to define keys. Within the Dictionary Editor, you can place the expression in the external name field in the **Key Properties** dialog. The format of the external name is:

```
'FileName=T[Expression]'
```

Where `FileName` represents the name of the index file (which can contain a path and file extension), and `T` represents the type of the index. Valid types are: C = character, D = date, and N = numeric. If the type is D or N

then *Expression* can name only one field.

String expressions may use the '+' operator to concatenate multiple string arguments. Numeric expressions use the '+' or '-' operators with their conventional meanings. The maximum length of a Clipper expression is 250 characters.

The expression may refer to multiple fields in the record, and may contain xBase functions. Square brackets must enclose the expression. The currently supported functions appear below. If the driver encounters an unsupported Xbase function in a preexisting file, it posts error 76 'Invalid Index String' when the file is opened for keys and static indexes.

### Supported xBase Key Definition Functions

|                                       |                                                                                                                                                                                                                                              |
|---------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ALLTRIM( <i>string</i> )              | Removes leading and trailing spaces.                                                                                                                                                                                                         |
| CTOD( <i>string</i> )                 | Converts a string key to a date. The <i>string</i> must be in the format mm/dd/yy; the result takes the form 'yyyymmdd'. The yyyy element of the date defaults to the twentieth century. An invalid date results in a key containing blanks. |
| DELETED()                             | Returns TRUE if the record is deleted.                                                                                                                                                                                                       |
| DESCEND( <i>string date numeric</i> ) | Inverts the argument, and creates descending Clipper indexes.                                                                                                                                                                                |
| DTOC( <i>date</i> )                   | Converts a date key to string format 'mm/dd/yy'                                                                                                                                                                                              |
| DTOS( <i>date</i> )                   | Converts a date key to string format 'yyyymmdd'                                                                                                                                                                                              |
| FIXED( <i>float</i> )                 | Converts a float key to a numeric.                                                                                                                                                                                                           |
| FLOAT( <i>numeric</i> )               | Converts a numeric key to a float.                                                                                                                                                                                                           |
| IIF( <i>bool, val1, val2</i> )        | Returns val1 if the first parameter is TRUE, otherwise returns val2.                                                                                                                                                                         |
| LEFT( <i>string, n</i> )              | Returns the leftmost <i>n</i> characters of the string key as a string of length <i>n</i> .                                                                                                                                                  |
| LOWER( <i>string</i> )                | Converts a string key to lower case.                                                                                                                                                                                                         |
| LTRIM( <i>string</i> )                | Removes spaces from the left of a string.                                                                                                                                                                                                    |
| RECNO()                               | Returns the current record number.                                                                                                                                                                                                           |
| RIGHT( <i>string, n</i> )             | Returns the rightmost <i>n</i> characters of the string key as a string of length <i>n</i> .                                                                                                                                                 |
| RTRIM( <i>string</i> )                | Removes spaces from the right of a string.                                                                                                                                                                                                   |

STR(numeric [,length[, decimal places]])

Converts a numeric to a string. The length of the string and the number of decimal places are optional. The default string length is 10, and the number of decimal places is 0.

SUBSTR(string,offset,n) Returns a substring of the *string* key starting at *offset* and of *n* characters in length.

TRIM(string) Removes spaces from the right of a string (identical to RTRIM).

UPPER(string) Converts a string key to upper case.

VAL(string) Converts a string key to a numeric.

Copies the UNNAMED (the only table in the file) table from CUSTOMER.TPS (which has no password) to the CUSTOMER table in ORDERS.TPS which has the password acme96.

## Configurable **ERROR** Messages

All of TopSpeed's database drivers post error conditions and messages that can be accessed with the `ERRORCODE()`, `ERROR()`, `FILEERRORCODE()` and `FILEERROR()` procedures (see the *Language Reference*). The drivers post the codes immediately after each I/O (`OPEN`, `NEXT`, `GET`, `ADD`, `DELETE`, etc.) operation.

**Tip:** The `ERRORFILE()` procedure returns the file or table that produced the error.

See *Run Time Errors* in the *Language Reference* for information on the `ERRORCODE()` values and their corresponding `ERROR()` message text. The `ERROR()` text is configurable using the environment file (`CLAMSG`) and the `LOCALE` procedure. See *Internationalization*, `CLAMSG`, and `LOCALE` in the *Language Reference* for more information.

In addition, Clipper driver `FILEERROR()` message text is also configurable using the environment file (`CLAMSG`) and the `LOCALE` procedure. However, the `CLAMSG` value required to change the message text is equal to the `FILEERRORCODE()` value + 4000. Following is a list of the `FILEERROR()` text, the `FILEERRORCODE()` values, and the `CLAMSG` values required to change the `FILEERROR()` text:

| <code>FILEERRORCODE</code> | <code>CLAMSG</code> | <code>FILEERROR</code>            |
|----------------------------|---------------------|-----------------------------------|
| 1                          | 4001                | file write failure                |
| 2                          | 4002                | file read failure                 |
| 3                          | 4003                | memory allocation error           |
| 4                          | 4004                | file pointer reposition failed    |
| 5                          | 4005                | file not found                    |
| 6                          | 4006                | file corrupted                    |
| 7                          | 4007                | bad user specified key expression |
| 8                          | 4008                | no handles available              |
| 9                          | 4009                | no index pages loaded             |
| 10                         | 4010                | index page was not loaded         |
| 11                         | 4011                | file close failure                |
| 12                         | 4012                | invalid command                   |
| 13                         | 4013                | invalid handle number             |
| 14                         | 4014                | invalid filename                  |
| 15                         | 4015                | invalid date                      |
| 16                         | 4016                | invalid time                      |
| 17                         | 4017                | file not in memo format           |
| 18                         | 4018                | invalid Clipper version           |
| 19                         | 4019                | file header length error          |
| 20                         | 4020                | last file change date in error    |
| 21                         | 4021                | parameter address NULL            |
| 22                         | 4022                | invalid key type                  |
| 23                         | 4023                | invalid key length                |

## FILEERRORCODE CLAMSG

## FILEERROR

|    |      |                                          |
|----|------|------------------------------------------|
| 24 | 4024 | item length incorrect                    |
| 24 | 4025 | invalid root page                        |
| 24 | 4026 | bad maximum number of keys per page      |
| 24 | 4027 | invalid number of fields                 |
| 24 | 4028 | field name invalid                       |
| 24 | 4029 | bad field length                         |
| 24 | 4030 | decimal places parameter invalid         |
| 24 | 4031 | invalid field type                       |
| 24 | 4032 | invalid record length                    |
| 24 | 4033 | bad data                                 |
| 24 | 4034 | memo soft line length invalid            |
| 24 | 4035 | MDX flag in DBF file invalid             |
| 24 | 4036 | file open for reading only               |
| 24 | 4037 | file locking violation                   |
| 24 | 4038 | sharing buffer overflow                  |
| 24 | 4039 | path not found                           |
| 24 | 4040 | access to file denied                    |
| 24 | 4041 | invalid access code                      |
| 24 | 4042 | file must be locked first                |
| 24 | 4043 | diskette changed                         |
| 24 | 4044 | bad minimum number of keys per page      |
| 24 | 4045 | some files remain open                   |
| 24 | 4046 | could not open the file                  |
| 24 | 4047 | flush to disk failure                    |
| 24 | 4048 | invalid tag handle                       |
| 24 | 4049 | invalid page size in blocks              |
| 24 | 4050 | invalid tag name                         |
| 24 | 4051 | invalid page offset adder                |
| 24 | 4052 | invalid max number of tag table elements |
| 24 | 4053 | bad tag table element length             |
| 24 | 4054 | invalid tag count                        |
| 24 | 4055 | unknown key format switches              |
| 24 | 4056 | unknown switch error                     |
| 24 | 4057 | tag in use                               |
| 24 | 4058 | Windows GlobalLock error                 |
| 24 | 4059 | Windows Task lookup Failed               |
| 24 | 4060 | internal lock buffer overflow            |
| 24 | 4061 | internal lock buffer underflow           |

# 29 - dBASEIII DATABASE DRIVER

## Specifications

The dBase3 file driver is compatible with dBase III. The default data file extension is \*.DBF.

Keys and Indexes exist as separate files from the data file. Keys are dynamic—they automatically update as the data file changes. Indexes are static—they do not automatically update, but instead require the BUILD statement for updating. The default file extension for the index file is \*.NDX. International sort orders are supported.

The driver stores records as fixed length. It stores memo fields in a separate file. The memo file name takes the first eight characters of the File Label plus an extension of .DBT.

## Files

---

|                    |                                       |
|--------------------|---------------------------------------|
| <b>C5DB3L.LIB</b>  | Windows Static Link Library (16-bit)  |
| <b>C5DB3XL.LIB</b> | Windows Static Link Library (32-bit)  |
| <b>C5DB3.LIB</b>   | Windows Export Library (16-bit)       |
| <b>C5DB3X.LIB</b>  | Windows Export Library (32-bit)       |
| <b>C5DB3.DLL</b>   | Windows Dynamic Link Library (16-bit) |
| <b>C5DB3X.DLL</b>  | Windows Dynamic Link Library (32-bit) |

**Tip:** dBase III is probably the most common file format for PC database applications. These days, even desktop publishing programs can import dBase III compatible .DBF files. If the main task of your application is to export data files for other applications about which you know nothing, you should consider this format.

## Data Types

---

The xBase file format stores all data as ASCII strings. You may either specify STRING types with declared pictures for each field, or specify native Clarion types, which the driver converts automatically.

| <u>dBase data type</u> | <u>Clarion data type</u> | <u>STRING w/ picture</u> |
|------------------------|--------------------------|--------------------------|
| Date                   | DATE                     | STRING(@D12)             |
| *Numeric               | REAL                     | STRING(@N-_p.d)          |
| *Logical               | BYTE                     | STRING(1)                |
| Character              | STRING                   | STRING                   |
| *Memo                  | MEMO                     | MEMO                     |

If your application reads and writes to existing files, a pictured STRING will suffice. However, if your application *creates* a dBase III file, you may require additional information for these dBase III types:

- ❖ To create a numeric field in the Data Dictionary, choose the REAL data type. In the External Name field on the Attributes tab, specify '*NumericFieldName=N(Precision,DecimalPlaces)*' where *NumericFieldName* is the name of the field, *Precision* is the precision of the field and *DecimalPlaces* is the number of decimal places. With a REAL data type, you cannot access the Character or Places fields in the Field definition, you must specify those attributes with an expression in the External Name Field on the Attributes tab.

For example, if you want to create a field called Number with nine significant digits and two decimal places, enter 'Number=N(9,2)' in the External Name field on the Attributes tab of the Field properties in the Data Dictionary.

If you're hand coding a native Clarion data type, add the NAME attribute using the same syntax.

If you're hand coding a STRING with picture, STRING(@N-\_9.2), NAME('Number'), where *Number* is the field name.

- ❖ To create a logical field, using the data dictionary, choose the BYTE data type. There are no special steps; however, see the miscellaneous section for tips on reading the data from the field.

If you're hand coding a STRING with picture, add the NAME attribute: STRING(1), NAME('LogFld = L').

- ❖ To create a date field, using the data dictionary, choose the DATE data type, rather than LONG, which you usually use for the TopSpeed or Clarion file formats.
- ❖ MEMO field declarations require the a pointer field in the file's record structure. Declare the pointer field as a STRING(10) or a LONG. This field will be stored in the .DBF file containing the offset of the memo in the .DBT file. The MEMO declaration must have a NAME() attribute naming the pointer field. An example file declaration follows:

```
File FILE, DRIVER('dBase3')
Memo1 MEMO(200),NAME('Notes')
Memo2 MEMO(200),NAME('Text')
Rec RECORD
Mem1Ptr LONG,NAME('Notes')
Mem2Ptr STRING(10),NAME('Text')
END
END
```

**Tip:** Use the File Import Utility in the Clarion Dictionary Editor to define your files.

## File Specifications/Maximums

---

|                        |                                  |
|------------------------|----------------------------------|
| File Size:             | 2,000,000,000 bytes              |
| Records per File:      | 1,000,000,000                    |
| Record Size:           | 4,000 bytes                      |
| Field Size             |                                  |
| Character:             | 254 bytes                        |
| Date:                  | 8 bytes                          |
| Logical:               | 1 byte                           |
| Numeric:               | 20 bytes including decimal point |
| Memo:                  | 64K (see note)                   |
| Fields per Record:     | 128                              |
| Keys/Indexes per File: | No Limit                         |
| Key Sizes              |                                  |
| Character:             | 100 bytes                        |
| Numeric, Date:         | 8 bytes                          |
| Memo fields per File:  | Dependent on available memory    |
| Open Files:            | Operating system dependent       |

## Driver Strings

There are switches or “driver strings” you can set to control the way your application creates, reads, and writes files with a specific driver. Driver strings are simply messages or parameters that are sent to the file driver at run-time to control its behavior. See *Common Driver Features—Driver Strings* for an overview of these runtime Database Driver switches and parameters.

**Note:** Some driver strings have no effect after the file is open, so no SEND function syntax is listed for those strings. However, the SEND function syntax to return the value of the switch is listed for all driver strings.

The dBaseIII Driver supports the following Driver Strings:

## BUFFERS

```
DRIVER('DBASE3', '/BUFFERS = n')
[Status" =] SEND(file, 'BUFFERS [= n]')
```

Sets the size of the buffer used to read and write to the file, where the buffer size is (n \* 512 bytes). Use the /BUFFERS driver string to increase the buffer size if access is slow. Maximum buffer size is 65,504 in 16-bit and 4,294,967,264 in 32-bit. SEND returns the size of the buffer in bytes.

**Tip:** The default is three buffers of 1024 bytes each. Increasing the number of buffers will not increase performance when a file is shared by multiple users.

## RECOVER

```
DRIVER('DBASE3', '/RECOVER')
[Status" =] SEND(file, 'RECOVER')
```

Equivalent to the Xbase RECALL command, which recovers records marked for deletion. When using the dBaseIV driver, the DELETE statement flags a record as “inactive.” The driver does not remove the record until the PACK command is executed.

RECOVER is evaluated each time you open the file if you add the driver string to the data dictionary. When the driver recovers the records previously marked for deletion, you must manually rebuild keys and indexes with the BUILD statement.

## IGNORESTATUS

---

```
DRIVER('DBASE3', '/IGNORESTATUS = on | off ')
[Status" =] SEND(file, 'IGNORESTATUS [on | off]')
```

When set *on*, the driver does *not* skip deleted records when accessing the file with GET, NEXT, and PREVIOUS in file order. It also enables a PUT on a deleted or held record. IGNORESTATUS requires opening the file in exclusive mode. SEND returns the IGNORESTATUS setting (ON or OFF) in the form of a STRING(3).

## DELETED

---

```
[Status" =] SEND(file, 'DELETED')
```

For use only with the SEND command, when IGNORESTATUS is on. Returns the status of the current record. If deleted, the return string is "ON" and if not, "OFF."

## OMNIS

---

```
DRIVER('DBASE3', '/OMNIS')
SEND(file, 'OMNIS')
```

Specifies OMNIS file header and file delimiter compatibility. SEND is only valid when the file is closed; it returns nothing.

## Supported Attributes and Procedures

### File Attributes Supported

|                                                          |                 |
|----------------------------------------------------------|-----------------|
| CREATE .....                                             | Y               |
| DRIVER( <i>filetype</i> [, <i>driver string</i> ]) ..... | Y               |
| NAME .....                                               | Y               |
| ENCRYPT .....                                            | N               |
| OWNER( <i>password</i> ) .....                           | N               |
| RECLAIM .....                                            | N <sup>1</sup>  |
| PRE( <i>prefix</i> ) .....                               | Y               |
| BINDABLE .....                                           | Y               |
| THREAD .....                                             | Y <sup>12</sup> |
| EXTERNAL( <i>member</i> ) .....                          | Y               |
| DLL( <i>[flag]</i> ) .....                               | Y               |
| OEM .....                                                | Y               |

### File Structures Supported

|              |   |
|--------------|---|
| INDEX .....  | Y |
| KEY .....    | Y |
| MEMO .....   | Y |
| BLOB .....   | N |
| RECORD ..... | Y |

### Index, Key, Memo Attributes Supported

|                             |                |
|-----------------------------|----------------|
| BINARY .....                | N              |
| DUP .....                   | Y <sup>2</sup> |
| NOCASE .....                | Y              |
| OPT .....                   | N              |
| PRIMARY .....               | Y              |
| NAME .....                  | Y              |
| Ascending Components .....  | Y              |
| Descending Components ..... | Y              |
| Mixed Components .....      | N              |

### Field Attributes Supported

|            |   |
|------------|---|
| DIM .....  | N |
| OVER ..... | Y |
| NAME ..... | Y |

### File Procedures Supported

|                                                 |                |
|-------------------------------------------------|----------------|
| BOF( <i>file</i> ) .....                        | Y <sup>8</sup> |
| BUFFER( <i>file</i> ) .....                     | N              |
| BUILD( <i>file</i> ) .....                      | Y              |
| BUILD( <i>key</i> ) .....                       | Y              |
| BUILD( <i>index</i> ) .....                     | Y              |
| BUILD( <i>index, components</i> ) .....         | Y <sup>3</sup> |
| BUILD( <i>index, components, filter</i> ) ..... | N              |
| BYTES( <i>file</i> ) .....                      | N              |
| CLOSE( <i>file</i> ) .....                      | Y              |
| COPY( <i>file, new file</i> ) .....             | Y <sup>4</sup> |
| CREATE( <i>file</i> ) .....                     | Y              |
| DUPLICATE( <i>file</i> ) .....                  | Y              |
| DUPLICATE( <i>key</i> ) .....                   | Y              |
| EMPTY( <i>file</i> ) .....                      | Y              |
| EOF( <i>file</i> ) .....                        | Y <sup>8</sup> |
| FLUSH( <i>file</i> ) .....                      | Y              |
| LOCK( <i>file</i> ) .....                       | N              |
| NAME( <i>label</i> ) .....                      | Y              |
| OPEN( <i>file, access mode</i> ) .....          | Y <sup>5</sup> |

|                                         |                 |
|-----------------------------------------|-----------------|
| PACK( <i>file</i> ) .....               | Y               |
| POINTER( <i>file</i> ) .....            | Y <sup>9</sup>  |
| POINTER( <i>key</i> ) .....             | Y <sup>9</sup>  |
| POSITION( <i>file</i> ) .....           | Y <sup>10</sup> |
| POSITION( <i>key</i> ) .....            | Y <sup>10</sup> |
| RECORDS( <i>file</i> ) .....            | Y <sup>11</sup> |
| RECORDS( <i>key</i> ) .....             | Y <sup>11</sup> |
| REMOVE( <i>file</i> ) .....             | Y               |
| RENAME( <i>file, new file</i> ) .....   | Y <sup>4</sup>  |
| SEND( <i>file, message</i> ) .....      | Y               |
| SHARE( <i>file, access mode</i> ) ..... | Y <sup>5</sup>  |
| STATUS( <i>file</i> ) .....             | Y               |
| STREAM( <i>file</i> ) .....             | Y               |
| UNLOCK( <i>file</i> ) .....             | N               |

### Record Access Supported

|                                               |                |
|-----------------------------------------------|----------------|
| ADD( <i>file</i> ) .....                      | Y <sup>6</sup> |
| ADD( <i>file, length</i> ) .....              | N              |
| APPEND( <i>file</i> ) .....                   | Y <sup>6</sup> |
| APPEND( <i>file, length</i> ) .....           | N              |
| DELETE( <i>file</i> ) .....                   | Y <sup>1</sup> |
| GET( <i>file, key</i> ) .....                 | Y              |
| GET( <i>file, filepointer</i> ) .....         | Y              |
| GET( <i>file, filepointer, length</i> ) ..... | N              |
| GET( <i>key, keypointer</i> ) .....           | Y              |
| HOLD( <i>file</i> ) .....                     | Y <sup>7</sup> |
| NEXT( <i>file</i> ) .....                     | Y              |
| NOMEMO( <i>file</i> ) .....                   | Y              |
| PREVIOUS( <i>file</i> ) .....                 | Y              |
| PUT( <i>file</i> ) .....                      | Y              |
| PUT( <i>file, filepointer</i> ) .....         | Y              |
| PUT( <i>file, filepointer, length</i> ) ..... | N              |
| RELEASE( <i>file</i> ) .....                  | Y              |
| REGET( <i>file, string</i> ) .....            | Y              |
| REGET( <i>key, string</i> ) .....             | Y              |
| RESET( <i>file, string</i> ) .....            | Y              |
| RESET( <i>key, string</i> ) .....             | Y              |
| SET( <i>file</i> ) .....                      | Y              |
| SET( <i>file, key</i> ) .....                 | Y              |
| SET( <i>file, filepointer</i> ) .....         | Y              |
| SET( <i>key</i> ) .....                       | Y              |
| SET( <i>key, key</i> ) .....                  | Y              |
| SET( <i>key, keypointer</i> ) .....           | Y              |
| SET( <i>key, key, filepointer</i> ) .....     | Y              |
| SKIP( <i>file, count</i> ) .....              | Y              |
| WATCH( <i>file</i> ) .....                    | Y              |

### Transaction Processing Supported<sup>13</sup>

|                                                 |   |
|-------------------------------------------------|---|
| LOGOUT( <i>timeout, file, ..., file</i> ) ..... | N |
| COMMIT .....                                    | N |
| ROLLBACK .....                                  | N |

### Null Data Processing Supported

|                                  |   |
|----------------------------------|---|
| NULL( <i>field</i> ) .....       | N |
| SETNULL( <i>field</i> ) .....    | N |
| SETNONNULL( <i>field</i> ) ..... | N |

## Notes

- 1 When the driver deletes a record from a dBase III database, the record is not physically removed, instead the driver marks it inactive. Memo fields are not physically removed from the memo file, however they cannot be retrieved if they refer to an inactive record. Key values *are* removed from the index files. To remove records and memo fields permanently, execute a PACK(file).

**Tip:** To those programmers familiar with dBase III, this driver processes deleted records consistent with the way dBase III processes them after the SET DELETED ON command is issued. Records marked for deletion are ignored from processing by executable code statements, but remain in the data file.

- 2 dBase III does not support any form of unique index. So the DUP attribute must be on all keys.
- 3 When building dynamic indexes, the *components* may take one of two forms:

```
BUILD(DynNdx, '+Pre:FLD1, -Pre:FLD2')
```

This form specifies the names of the fields on which to build the index. The field names must appear as specified in the fields' NAME() attribute if supplied, or must be the label name. A prefix may be used for compatibility with Clarion conventions but is ignored.

```
BUILD(DynNdx, 'T[Expression]')
```

This form specifies the type and Expression used to build an index—see *Miscellaneous—Key Definition* below.

- 4 These commands copy data and memo files using *newfile*, which may specify a new file name or directory. Key or index files are copied if the *newfile* is a subdirectory specification. To copy an index file to a new file, use a special form of the copy or rename command:

```
COPY(file, '<index>|<newfile>')
```

This returns *File Not Found* if an invalid index is passed. The COPY command assumes a default extension of ".NDX" for both the source and the target file names if none is specified. If you require a file name without an extension, terminate the name with a period. Given the file structure:

```
Clar2 FILE,CREATE,DRIVER('dBase3'),PRE(CL2)
NumKey KEY(+CL2:Num),DUP
StrKey KEY(+CL2:Str1)
StrKey2 KEY(+CL2:Str2)
AMemo MEMO(100), NAME('mem')
Record RECORD
Num STRING(@n_.9.2)
STR1 STRING(2)
```

```
STR2 STRING(2)
Mem STRING(10)
```

The following commands copy this file definition to A:

```
COPY(Clar2, 'A:\CLAR2')
COPY(Clar2, 'StrKey|A:\STRKEY')
COPY(Clar2, 'StrKey2|A:\STRKEY2')
COPY(Clar2, 'NumKey|A:\NUMKEY')
```

After these calls, the following files would exist on drive A: CLAR2.DBF, CLAR2.DBT, STRKEY.NDX, STRKEY2.NDX, and NUMKEY.NDX.

- 5 You do not need SHARE (or VSHARE) in any environment (for example, Novell) that supplies file locking as part of the operating system.
- 6 The ADD statement tests for duplicate keys before modifying the data file or its associated KEY files. Consequently it is slower than APPEND which performs no checks and does not update KEYS. When adding large amounts of data to a database use APPEND...BUILD in preference to ADD.
- 7 dBase III performs record locking by locking the entire record within the data file. This prevents read access to other processes. Therefore we recommend minimizing the amount of time for which a record is held.
- 8 Although the driver supports these functions, we do not recommend their use. They must physically access the files and add overhead. Instead, test the value returned by ERRORCODE() after each sequential access. NEXT or PREVIOUS post Error 33 (Record Not Available) if an attempt is made to access a record beyond the end or beginning of the file.
- 9 There is no distinction between file pointers and key pointers; they both return the record number for any given record.
- 10 POSITION(file) returns a STRING(12). POSITION(key) returns a STRING the size of the key fields + 4 bytes.
- 11 Under dBase III, the RECORDS() function reports the same number of records for the data file and its keys and indexes. Usually there will be no difference in the number of records *unless* the INDEX is out of date. Because the DELETE statement does not physically remove records, *the number of records reported by the RECORDS() function includes inactive records*. Exercise care when using this function.
- 12 THREADED files consume additional file handles for each thread that accesses the file.
- 13 You should clear the **Enclose RI code in transaction frame** box on the **File Control** tab of the **Global Properties** dialog when using this driver with template generated applications.

## Other

### Boolean Evaluation

- ❖ dBase III allows a logical field to accept one of nine possible values (y,Y,n,N,t,T,f,F or a space character). The space character is neither true nor false. When using a logical field from a preexisting database in a logical expression, account for all these possibilities. Remember that when a STRING field is used as an expression, it is true if it contains any data and false if it is equal to zero or blank. Therefore, to evaluate a Logical field's truth, the expression should be true if the field contains any of the "true" characters (T,t,Y, or y). For example, if a Logical field were used to specify a product as taxable or nontaxable, the expression to evaluate its truth would be:

*(If Condition):*

```
Taxable='T' OR Taxable='t' OR Taxable='Y' OR Taxable='y'
```

### Large MEMOs

- ❖ Clarion supports MEMO fields up to a maximum of 64K. If you have an existing file which includes a memo greater than 64K, you can use the file but not modify the large MEMOs.
- ❖ You can determine when your application encounters a large MEMO by detecting when the memo pointer variable is non-blank, but the memo appears to be blank. Error 47 (Bad Record Declaration) is posted, and any modification to the MEMO field is ignored.

### Long Field Names

- ❖ dBase III supports a maximum of 10 characters in a field name. If you require more, use an External Name with 10 characters or less.

### International Sort Sequence

- ❖ The dBaseIII driver supports international sort orders, however, to maintain compatibility with dBaseIII's international sort order, remove the CLADIGRAPH= line from ..\CLARION4\BIN\CLARION4.ENV file.

### KEY Definitions

- ❖ dBase III supports the use of expressions to define keys. Within the Dictionary Editor, you can place the expression in the external name field in the *Key Properties* dialog. The general format of the external name is :

```
'FileName=T[Expression]'
```

Where `FileName` represents the name of the index file (which can contain

a path and file extension), and  $\tau$  represents the type of the index. Valid types are: C = character, D = date, and N = numeric. If the type is D or N then `Expression` can name only one field.

String expressions may use the '+' operator to concatenate multiple string arguments. Numeric expressions use the '+' or '-' operators with their conventional meanings. The maximum length of a dBase III expression is 250 characters.

The expression may refer to multiple fields in the record, and contain xBase functions. Square brackets must enclose the expression. The currently supported functions appear below. If the driver encounters an unsupported Xbase function in a preexisting file, it posts error 76 'Invalid Index String' when the file is opened for keys and static indexes.

### **Supported xBase Key Definition Functions**

|                                             |                                                                                                                                                                                                                                              |
|---------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ALLTRIM(string)                             | Removes leading and trailing spaces.                                                                                                                                                                                                         |
| CTOD(string)                                | Converts a string key to a date. The <i>string</i> must be in the format mm/dd/yy; the result takes the form 'yyyymmdd'. The yyyy element of the date defaults to the twentieth century. An invalid date results in a key containing blanks. |
| DELETED()                                   | Returns TRUE if the record is deleted.                                                                                                                                                                                                       |
| DTOC(date)                                  | Converts a date key to string format 'mm/dd/yy.'                                                                                                                                                                                             |
| DTOS(date)                                  | Converts a date key to string format 'yyyymmdd.'                                                                                                                                                                                             |
| FIXED(float)                                | Converts a float key to a numeric.                                                                                                                                                                                                           |
| FLOAT(numeric)                              | Converts a numeric key to a float.                                                                                                                                                                                                           |
| IIF(bool, val1, val2)                       | Returns val1 if the first parameter is TRUE, otherwise returns val2.                                                                                                                                                                         |
| LEFT(string, n)                             | Returns the leftmost <i>n</i> characters of the string key as a string of length <i>n</i> .                                                                                                                                                  |
| LOWER(string)                               | Converts a string key to lower case.                                                                                                                                                                                                         |
| LTRIM(string)                               | Removes spaces from the left of a string.                                                                                                                                                                                                    |
| RECNO()                                     | Returns the current record number.                                                                                                                                                                                                           |
| RIGHT(string, n)                            | Returns the rightmost <i>n</i> characters of the string key as a string of length <i>n</i> .                                                                                                                                                 |
| RTRIM(string)                               | Removes spaces from the right of a string.                                                                                                                                                                                                   |
| STR(numeric [,length [, decimal places] ] ) | Converts a numeric to a string. The length of the                                                                                                                                                                                            |

string and the number of decimal places are optional. The default string length is 10, and the number of decimal places is 0.

`SUBSTR(string,offset,n)` Returns a substring of the *string* key starting at *offset* and of *n* characters in length.

`TRIM(string)` Removes spaces from the right of a string (identical to `RTRIM`).

`UPPER(string)` Converts a string key to upper case.

`VAL(string)` Converts a string key to a numeric.

## Configurable **ERROR** Messages

All of TopSpeed's database drivers post error conditions and messages that can be accessed with the `ERRORCODE()`, `ERROR()`, `FILEERRORCODE()` and `FILEERROR()` procedures (see the *Language Reference*). The drivers post the codes immediately after each I/O (`OPEN`, `NEXT`, `GET`, `ADD`, `DELETE`, etc.) operation.

**Tip:** The `ERRORFILE()` procedure returns the file or table that produced the error.

See *Run Time Errors* in the *Language Reference* for information on the `ERRORCODE()` values and their corresponding `ERROR()` message text. The `ERROR()` text is configurable using the environment file (`CLAMSG`) and the `LOCALE` procedure. See *Internationalization*, `CLAMSG`, and `LOCALE` in the *Language Reference* for more information.

In addition, dBaseIII driver `FILEERROR()` message text is also configurable using the environment file (`CLAMSG`) and the `LOCALE` procedure. However, the `CLAMSG` value required to change the message text is equal to the `FILEERRORCODE()` value + 4000. Following is a list of the `FILEERROR()` text, the `FILEERRORCODE()` values, and the `CLAMSG` values required to change the `FILEERROR()` text:

| <code>FILEERRORCODE</code> | <code>CLAMSG</code> | <code>FILEERROR</code>            |
|----------------------------|---------------------|-----------------------------------|
| 1                          | 4001                | file write failure                |
| 2                          | 4002                | file read failure                 |
| 3                          | 4003                | memory allocation error           |
| 4                          | 4004                | file pointer reposition failed    |
| 5                          | 4005                | file not found                    |
| 6                          | 4006                | file corrupted                    |
| 7                          | 4007                | bad user specified key expression |
| 8                          | 4008                | no handles available              |
| 9                          | 4009                | no index pages loaded             |
| 10                         | 4010                | index page was not loaded         |
| 11                         | 4011                | file close failure                |
| 12                         | 4012                | invalid command                   |
| 13                         | 4013                | invalid handle number             |
| 14                         | 4014                | invalid filename                  |
| 15                         | 4015                | invalid date                      |
| 16                         | 4016                | invalid time                      |
| 17                         | 4017                | file not in memo format           |
| 18                         | 4018                | invalid dBaseIII version          |
| 19                         | 4019                | file header length error          |
| 20                         | 4020                | last file change date in error    |
| 21                         | 4021                | parameter address NULL            |
| 22                         | 4022                | invalid key type                  |
| 23                         | 4023                | invalid key length                |

**FILEERRORCODE CLAMSG****FILEERROR**

|    |      |                                          |
|----|------|------------------------------------------|
| 24 | 4024 | item length incorrect                    |
| 24 | 4025 | invalid root page                        |
| 24 | 4026 | bad maximum number of keys per page      |
| 24 | 4027 | invalid number of fields                 |
| 24 | 4028 | field name invalid                       |
| 24 | 4029 | bad field length                         |
| 24 | 4030 | decimal places parameter invalid         |
| 24 | 4031 | invalid field type                       |
| 24 | 4032 | invalid record length                    |
| 24 | 4033 | bad data                                 |
| 24 | 4034 | memo soft line length invalid            |
| 24 | 4035 | MDX flag in DBF file invalid             |
| 24 | 4036 | file open for reading only               |
| 24 | 4037 | file locking violation                   |
| 24 | 4038 | sharing buffer overflow                  |
| 24 | 4039 | path not found                           |
| 24 | 4040 | access to file denied                    |
| 24 | 4041 | invalid access code                      |
| 24 | 4042 | file must be locked first                |
| 24 | 4043 | diskette changed                         |
| 24 | 4044 | bad minimum number of keys per page      |
| 24 | 4045 | some files remain open                   |
| 24 | 4046 | could not open the file                  |
| 24 | 4047 | flush to disk failure                    |
| 24 | 4048 | invalid tag handle                       |
| 24 | 4049 | invalid page size in blocks              |
| 24 | 4050 | invalid tag name                         |
| 24 | 4051 | invalid page offset adder                |
| 24 | 4052 | invalid max number of tag table elements |
| 24 | 4053 | bad tag table element length             |
| 24 | 4054 | invalid tag count                        |
| 24 | 4055 | unknown key format switches              |
| 24 | 4056 | unknown switch error                     |
| 24 | 4057 | tag in use                               |
| 24 | 4058 | Windows GlobalLock error                 |
| 24 | 4059 | Windows Task lookup Failed               |
| 24 | 4060 | internal lock buffer overflow            |
| 24 | 4061 | internal lock buffer underflow           |



# 30 - dBASE IV DATABASE DRIVER

## Specifications

The dBase4 file driver is compatible with dBase IV. The default data file extension is \*.DBF.

Keys and Indexes exist as separate files from the data file. Keys are dynamic—they automatically update as the data file changes. Indexes are static—they do not automatically update, but instead require the BUILD statement for updating. The default file extension for the index file is \*.NDX.

dBase IV supports multiple index files, whose extension is \*.MDX. The miscellaneous section describes procedures for using .MDX files.

The driver stores records as fixed length. It stores memo fields in a separate file. The memo file name takes the first eight characters of the File Label plus an extension of .DBT.

## Files

---

|                    |                                       |
|--------------------|---------------------------------------|
| <b>C5DB4L.LIB</b>  | Windows Static Link Library (16-bit)  |
| <b>C5DB4XL.LIB</b> | Windows Static Link Library (32-bit)  |
| <b>C5DB4.LIB</b>   | Windows Export Library (16-bit)       |
| <b>C5DB4X.LIB</b>  | Windows Export Library (32-bit)       |
| <b>C5DB4.DLL</b>   | Windows Dynamic Link Library (16-bit) |
| <b>C5DB4X.DLL</b>  | Windows Dynamic Link Library (32-bit) |

**Tip:** dBase IV was never as widely adopted as dBase III. Choose this driver only when you must share data with an end-user using dBase IV.

## Data Types

---

The xBase file format stores all data as ASCII strings. You may either specify STRING types with declared pictures for each field, or specify native Clarion types, which the driver converts automatically.

| <u>dBase data type</u> | <u>Clarion data type</u> | <u>STRING w/ picture</u> |
|------------------------|--------------------------|--------------------------|
| Date                   | DATE                     | STRING(@D12)             |
| *Numeric               | REAL                     | STRING(@N-_p.d)          |
| *Logical               | BYTE                     | STRING(1)                |
| Character              | STRING                   | STRING                   |
| *Memo                  | MEMO                     | MEMO                     |

If your application reads and writes to existing files, a pictured STRING will suffice. However, if your application *creates* a dBase IV file, you may require additional information for these dBase IV types:

- ❖ To create a numeric field in the Data Dictionary, choose the REAL data type. In the External Name field on the Attributes tab, specify '*NumericFieldName=N(Precision,DecimalPlaces)*' where *NumericFieldName* is the name of the field, *Precision* is the precision of the field and *DecimalPlaces* is the number of decimal places. With a REAL data type, you cannot access the Character or Places fields in the Field definition, you must specify those attributes with an expression in the External Name Field on the Attributes tab.

For example, if you want to create a field called Number with nine significant digits and two decimal places, enter 'Number=N(9,2)' in the External Name field on the Attributes tab of the Field properties in the Data Dictionary.

If you're hand coding a native Clarion data type, add the NAME attribute using the same syntax.

If you're hand coding a STRING with picture, STRING(@N-9.2), NAME('Number'), where *Number* is the field name.

- ❖ To create a logical field, using the data dictionary, choose the BYTE data type. There are no special steps; however, see the miscellaneous section for tips on reading the data from the field.

If you're hand coding a STRING with picture, add the NAME attribute: STRING(1), NAME('LogFld = L').

- ❖ To create a date field, using the data dictionary, choose the DATE data type, rather than LONG, which you usually use for the TopSpeed or Clarion file formats.
- ❖ MEMO field declarations require the a pointer field in the file's record structure. Declare the pointer field as a STRING(10) or a LONG. This field will be stored in the .DBF file containing the offset of the memo in the .DBT file. The MEMO declaration must have a NAME() attribute naming the pointer field. An example file declaration follows:

```
File FILE, DRIVER('dBase4')
Memo1 MEMO(200),NAME('Notes')
Memo2 MEMO(200),NAME('Text')
Rec RECORD
Mem1Ptr LONG,NAME('Notes')
Mem2Ptr STRING(10),NAME('Text')
END
END
```

**Tip:** Use the File Import Utility in the Clarion Dictionary Editor to define your files.

## File Specifications/Maximums

---

|                        |                                  |
|------------------------|----------------------------------|
| File Size:             | 2,000,000,000 bytes              |
| Records per File:      | 1,000,000,000                    |
| Record Size:           | 4,000 bytes                      |
| Field Size             |                                  |
| Character:             | 254 bytes                        |
| Date:                  | 8 bytes                          |
| Logical:               | 1 byte                           |
| Numeric:               | 20 bytes including decimal point |
| Float:                 | 20 bytes including decimal point |
| Memo:                  | 64K (see note)                   |
| Fields per Record:     | 512                              |
| Keys/Indexes per File: |                                  |
| .NDX:                  | No Limit                         |
| .MDX                   | 47 tags per .MDX files           |
| Key Sizes              |                                  |
| Character:             | 100 bytes                        |
| Numeric, Date:         | 8 bytes                          |
| Memo fields per File:  | Dependent on available memory    |
| Open Files:            | Operating system dependent       |

## Driver Strings

There are switches or “driver strings” you can set to control the way your application creates, reads, and writes files with a specific driver. Driver strings are simply messages or parameters that are sent to the file driver at run-time to control its behavior. See *Common Driver Features—Driver Strings* for an overview of these runtime Database Driver switches and parameters.

**Note:** Some driver strings have no effect after the file is open, so no SEND function syntax is listed for those strings. However, the SEND function syntax to return the value of the switch is listed for all driver strings.

The dBaseIV Driver supports the following Driver Strings:

## BUFFERS

```
DRIVER('DBASE4', '/BUFFERS = n')
[Status" =] SEND(file, 'BUFFERS [= n]')
```

Sets the size of the buffer used to read and write to the file, where the buffer size is (n \* 512 bytes). Use the /BUFFERS driver string to increase the buffer size if access is slow. Maximum buffer size is 65,504 in 16-bit and 4,294,967,264 in 32-bit. SEND returns the size of the buffer in bytes.

**Tip:** The default is three buffers of 1024 bytes each. Increasing the number of buffers will not increase performance when a file is shared by multiple users.

## RECOVER

```
DRIVER('DBASE4', '/RECOVER')
[Status" =] SEND(file, 'RECOVER')
```

Equivalent to the Xbase RECALL command, which recovers records marked for deletion. When using the dBaseIV driver, the DELETE statement flags a record as “inactive.” The driver does not remove the record until the PACK command is executed.

RECOVER is evaluated each time you open the file if you add the driver string to the data dictionary. When the driver recovers the records previously marked for deletion, you must manually rebuild keys and indexes with the BUILD statement.

## IGNORESTATUS

---

```
DRIVER('DBASE4', '/IGNORESTATUS = on | off ')
[Status" =] SEND(file, 'IGNORESTATUS [on | off]')
```

When set *on*, the driver does *not* skip deleted records when accessing the file with GET, NEXT, and PREVIOUS in file order. It also enables a PUT on a deleted or held record. IGNORESTATUS requires opening the file in exclusive mode. SEND returns the IGNORESTATUS setting (ON or OFF) in the form of a STRING(3).

## DELETED

---

```
[Status" =] SEND(file, 'DELETED')
```

For use only with the SEND command, when IGNORESTATUS is on. Returns the status of the current record. If deleted, the return string is "ON" and if not, "OFF."

## Supported Attributes and Procedures

### File Attributes Supported

|                                                          |                 |
|----------------------------------------------------------|-----------------|
| CREATE .....                                             | Y               |
| DRIVER( <i>filetype</i> [, <i>driver string</i> ]) ..... | Y               |
| NAME .....                                               | Y               |
| ENCRYPT .....                                            | N               |
| OWNER( <i>password</i> ) .....                           | N               |
| RECLAIM .....                                            | N <sup>1</sup>  |
| PRE( <i>prefix</i> ) .....                               | Y               |
| BINDABLE .....                                           | Y               |
| THREAD .....                                             | Y <sup>12</sup> |
| EXTERNAL( <i>member</i> ) .....                          | Y               |
| DLL( <i>flag</i> ) .....                                 | Y               |
| OEM .....                                                | Y               |

### File Structures Supported

|              |   |
|--------------|---|
| INDEX .....  | Y |
| KEY .....    | Y |
| MEMO .....   | Y |
| BLOB .....   | N |
| RECORD ..... | Y |

### Index, Key, Memo Attributes Supported

|                             |                |
|-----------------------------|----------------|
| BINARY .....                | N              |
| DUP .....                   | Y <sup>2</sup> |
| NOCASE .....                | Y              |
| OPT .....                   | N              |
| PRIMARY .....               | Y              |
| NAME .....                  | Y              |
| Ascending Components .....  | Y              |
| Descending Components ..... | Y              |
| Mixed Components .....      | N              |

### Field Attributes Supported

|            |   |
|------------|---|
| DIM .....  | N |
| OVER ..... | Y |
| NAME ..... | Y |

### File Procedures Supported

|                                                 |                |
|-------------------------------------------------|----------------|
| BOF( <i>file</i> ) .....                        | Y <sup>8</sup> |
| BUFFER( <i>file</i> ) .....                     | N              |
| BUILD( <i>file</i> ) .....                      | Y              |
| BUILD( <i>key</i> ) .....                       | Y              |
| BUILD( <i>index</i> ) .....                     | Y              |
| BUILD( <i>index, components</i> ) .....         | Y <sup>3</sup> |
| BUILD( <i>index, components, filter</i> ) ..... | N              |
| BYTES( <i>file</i> ) .....                      | N              |
| CLOSE( <i>file</i> ) .....                      | Y              |
| COPY( <i>file, new file</i> ) .....             | Y <sup>4</sup> |
| CREATE( <i>file</i> ) .....                     | Y              |
| DUPLICATE( <i>file</i> ) .....                  | Y              |
| DUPLICATE( <i>key</i> ) .....                   | Y              |
| EMPTY( <i>file</i> ) .....                      | Y              |
| EOF( <i>file</i> ) .....                        | Y <sup>8</sup> |
| FLUSH( <i>file</i> ) .....                      | Y              |
| LOCK( <i>file</i> ) .....                       | N              |
| NAME( <i>label</i> ) .....                      | Y              |
| OPEN( <i>file, access mode</i> ) .....          | Y <sup>5</sup> |

|                                         |                 |
|-----------------------------------------|-----------------|
| PACK( <i>file</i> ) .....               | Y               |
| POINTER( <i>file</i> ) .....            | Y <sup>9</sup>  |
| POINTER( <i>key</i> ) .....             | Y <sup>9</sup>  |
| POSITION( <i>file</i> ) .....           | Y <sup>10</sup> |
| POSITION( <i>key</i> ) .....            | Y <sup>10</sup> |
| RECORDS( <i>file</i> ) .....            | Y <sup>11</sup> |
| RECORDS( <i>key</i> ) .....             | Y <sup>11</sup> |
| REMOVE( <i>file</i> ) .....             | Y               |
| RENAME( <i>file, new file</i> ) .....   | Y <sup>4</sup>  |
| SEND( <i>file, message</i> ) .....      | Y               |
| SHARE( <i>file, access mode</i> ) ..... | Y <sup>5</sup>  |
| STATUS( <i>file</i> ) .....             | Y               |
| STREAM( <i>file</i> ) .....             | Y               |
| UNLOCK( <i>file</i> ) .....             | N               |

### Record Access Supported

|                                               |                |
|-----------------------------------------------|----------------|
| ADD( <i>file</i> ) .....                      | Y <sup>6</sup> |
| ADD( <i>file, length</i> ) .....              | N              |
| APPEND( <i>file</i> ) .....                   | Y <sup>6</sup> |
| APPEND( <i>file, length</i> ) .....           | N              |
| DELETE( <i>file</i> ) .....                   | Y <sup>1</sup> |
| GET( <i>file, key</i> ) .....                 | Y              |
| GET( <i>file, filepointer</i> ) .....         | Y              |
| GET( <i>file, filepointer, length</i> ) ..... | N              |
| GET( <i>key, keypointer</i> ) .....           | Y              |
| HOLD( <i>file</i> ) .....                     | Y <sup>7</sup> |
| NEXT( <i>file</i> ) .....                     | Y              |
| NOMEMO( <i>file</i> ) .....                   | Y              |
| PREVIOUS( <i>file</i> ) .....                 | Y              |
| PUT( <i>file</i> ) .....                      | Y              |
| PUT( <i>file, filepointer</i> ) .....         | Y              |
| PUT( <i>file, filepointer, length</i> ) ..... | N              |
| RELEASE( <i>file</i> ) .....                  | Y              |
| REGET( <i>file, string</i> ) .....            | Y              |
| REGET( <i>key, string</i> ) .....             | Y              |
| RESET( <i>file, string</i> ) .....            | Y              |
| RESET( <i>key, string</i> ) .....             | Y              |
| SET( <i>file</i> ) .....                      | Y              |
| SET( <i>file, key</i> ) .....                 | Y              |
| SET( <i>file, filepointer</i> ) .....         | Y              |
| SET( <i>key</i> ) .....                       | Y              |
| SET( <i>key, key</i> ) .....                  | Y              |
| SET( <i>key, keypointer</i> ) .....           | Y              |
| SET( <i>key, key, filepointer</i> ) .....     | Y              |
| SKIP( <i>file, count</i> ) .....              | Y              |
| WATCH( <i>file</i> ) .....                    | Y              |

### Transaction Processing Supported<sup>13</sup>

|                                                 |   |
|-------------------------------------------------|---|
| LOGOUT( <i>timeout, file, ..., file</i> ) ..... | N |
| COMMIT .....                                    | N |
| ROLLBACK .....                                  | N |

### Null Data Processing Supported

|                                  |   |
|----------------------------------|---|
| NULL( <i>field</i> ) .....       | N |
| SETNULL( <i>field</i> ) .....    | N |
| SETNONNULL( <i>field</i> ) ..... | N |

## Notes

---

- 1 When the driver deletes a record from a dBase IV database, the record is not physically removed, instead the driver marks it inactive. Memo fields are not physically removed from the memo file, however they cannot be retrieved if they refer to an inactive record. Key values *are* removed from the index files. To remove records and memo fields permanently, execute a PACK(file).

**Tip:** To those programmers familiar with dBase IV, this driver processes deleted records consistent with the way dBase IV processes them after the SET DELETED ON command is issued. Records marked for deletion are ignored from processing by executable code statements, but remain in the data file.

- 2 In dBase IV it is legal to enter multiple records with duplicates of the unique key components. However, only the first of these records is indexed. So processing in key order only shows this first record. If you delete a record, then enter a new record with the same key value, the key file continues to point at the deleted record rather than the new record. In this situation, the dBase IV file driver driver changes the key file to point at the active record rather than the deleted record. This means that if you use a dBase IV program to delete a unique record, then insert a duplicate of this record, the new record is invisible when processing in key order until a pack is done. If you do the same process in a Clarion program, the new record is visible when processing in key order.
- 3 When building dynamic indexes, the *components* may take one of two forms:

```
BUILD(DynNdx, '+Pre:FLD1, -Pre:FLD2')
```

This form specifies the names of the fields on which to build the index. The field names must appear as specified in the fields' NAME() attribute if supplied, or must be the label name. A prefix may be used for compatibility with Clarion conventions but is ignored.

```
BUILD(DynNdx, 'T[Expression]')
```

This form specifies the type and Expression used to build an index—see *Miscellaneous—Key Definition* below.

- 4 These commands copy data and memo files using *newfile*, which may specify a new file name or directory. Key or index files are copied if the *newfile* is a subdirectory specification. To copy an index file to a new file, use a special form of the copy command:

```
COPY(file, '<index>|<newfile>')
```

This returns *File Not Found* if an invalid index is passed. The COPY command assumes a default extension of .NDX for both the source and the target file names if none is specified. If you require a file name

without an extension, terminate the name with a period. Given the file structure:

```
CLar2 FILE,CREATE,DRIVER('dBase4'),PRE(CL2)
NumKey KEY(+CL2:Num),DUP
StrKey KEY(+CL2:Str1)
StrKey2 KEY(+CL2:Str2)
AMemo MEMO(100),NAME('mem')
Record RECORD
Num STRING(@n-_9.2)
STR1 STRING(2)
STR2 STRING(2)
Mem STRING(10)
. . .
```

The following commands copy this file definition to A:

```
COPY(Clar2,'A:\CLAR2')
COPY(Clar2,'StrKey|A:\STRKEY')
COPY(Clar2,'StrKey2|A:\STRKEY2')
COPY(Clar2,'NumKey|A:\NUMKEY')
```

After these calls, the following files would exist on drive A: CLAR2.DBF, CLAR2.DBT, STRKEY.NDX, STRKEY2.NDX, and NUMKEY.NDX.

- 5 You do not need SHARE (or VSHARE) in any environment (for example, Novell) that supplies file locking as part of the operating system.
- 6 The ADD statement tests for duplicate keys before modifying the data file or its associated KEY files. Consequently it is slower than APPEND which performs no checks and does not update KEYS. When adding large amounts of data to a database use APPEND...BUILD in preference to ADD.
- 7 dBase IV performs record locking by locking the entire record within the data file. This prevents read access to other processes. Therefore we recommend minimizing the amount of time for which a record is held.
- 8 Although the driver supports these functions, we do not recommend their use. They must physically access the files and add overhead. Instead, test the value returned by ERRORCODE() after each sequential access. NEXT or PREVIOUS post Error 33 (Record Not Available) if an attempt is made to access a record beyond the end or beginning of the file.
- 9 There is no distinction between file pointers and key pointers; they both return the record number for any given record.
- 10 POSITION(file) returns a STRING(12). POSITION(key) returns a STRING containing the size of the key fields + 4 bytes.
- 11 Under dBase IV, the RECORDS() function reports the same number of records for the data file and its keys and indexes. Usually there will be no difference in the number of records *unless* the INDEX is out of date. Because the DELETE statement does not physically remove records, *the*

*number of records reported by the RECORDS() function includes inactive records.* Exercise care when using this function. The field names must appear as specified in the fields' NAME() attribute if supplied, or must be the label name. A prefix may be used for compatibility with the Clarion conventions but is ignored.

- 12 THREADED files consume additional file handles for each thread that accesses the file.
- 13 You should clear the **Enclose RI code in transaction frame** box on the **File Control** tab of the **Global Properties** dialog when using this driver with template generated applications.

## Other

### International Sort Sequence

- ❖ dBase IV sorts as if there were no diacritics in a field, thus A is sorted the same as Ä. If two words are identical except for diacritic characters, then the words are sorted as though the diacritic character was greater than the normal character. For example Äa < Ab < Äb whereas a CLADIGRAPH of ÄAE will sort as Ab < Äa < Äb. Solution- if the same file is used in Clarion and dBase IV, issue a BUILD statement rebuild the keys before updating the file (reading the file causes no problems).

### Boolean Evaluation

- ❖ dBase IV allows a logical field to accept one of 11 possible values (1,0,y,Y,n,N,t,T,f,F or a space character). The space character is neither true nor false. When using a logical field from a preexisting database in a logical expression, account for all these possibilities. Remember that when a STRING field is used as an expression, it is true if it contains any data and false if it is equal to zero or blank. Therefore, to evaluate a Logical field's truth, the expression should be true if the field contains any of the "true" characters (T,t,Y, or y). For example, if a Logical field were used to specify a product as taxable or nontaxable, the expression to evaluate its truth would be:

*(If Condition):*

```
Taxable='1' OR Taxable='T' OR Taxable='t' OR Taxable='Y' OR Taxable='y'
```

### Large MEMOs

- ❖ Clarion supports MEMO fields up to a maximum of 64K. If you have an existing file which includes a memo greater than 64K, you can use the file but not modify the large MEMOs.
- ❖ You can determine when your application encounters a large MEMO by detecting when the memo pointer variable is non-blank, but the memo appears to be blank. Error 47 (Bad Record Declaration) is posted, and any modification to the MEMO field is ignored.

## Long Field Names

- ❖ dBase IV supports a maximum of 10 characters in a field name. If you require more, use an External Name with 10 characters or less.

## Key Definition

- ❖ dBase IV supports the use of expressions to define keys. Within the Dictionary Editor, you can place the expression in the external name field in the *Key Properties* dialog. The general format of the external name is :

```
'FileName=T[Expression]'
```

Where `FileName` represents the name of the index file (which can contain a path and file extension), and `T` represents the type of the index. Valid types are: C = character, D = date, and N = numeric. If the type is D or N then `Expression` can name only one field.

- ❖ Multiple-index (.MDX) files require the NAME() attribute on a KEY or INDEX to specify the storage type of the key and any expression used to generate the key values. The general format of the NAME() attribute on a KEY or INDEX is:

```
NAME('TagName|FileName[PageSize]=T[Expression],FOR[Expression]')
```

The following documents the parameters for the NAME() attribute:

|          |                                                                                                                                                                                                                                                                                                                                                                       |
|----------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| TagName  | Specifies the name of an index tag within a multiple index file. If omitted the driver creates a dBase IV style .NDX file using the name specified in FileName.                                                                                                                                                                                                       |
| FileName | Specifies the name of the index file, which may contain a path and extension.                                                                                                                                                                                                                                                                                         |
| PageSize | Specifies that when creating a .MDX file, (if a TagName is specified), a number in the range 2-32 specifying the number of 512-byte blocks in each index page. This value is only used when creating the file. If you specify multiple values with declarations for different tags in the same .MDX file, the largest value will be selected. The default value is 2. |
| T        | Specifies the type of index. Legal types are C = character, D = date, N = numeric. If the type is D or N then <i>Expression</i> may name <i>only one</i> field.                                                                                                                                                                                                       |

Expression Specifies an expression to generate the index. It may refer to multiple fields and invoke multiple xBase functions. The functions currently supported are listed below. Square brackets must enclose the expression.

Elements of the NAME() attribute may be omitted from the right. When specifying an Expression, you must also specify the type and name. If the Expression is omitted, the driver determines the Expression from the key fields when the file is created, or from the index file when opened.

If the type is omitted, the driver determines the index type from the first key component when the file is created, or from the index file when opened.

If the NAME() attribute is omitted altogether, the index file name is determined from the key label. The path defaults to the same location as the .DBF.

Tag names are limited to 9 characters in length. If the supplied name is too long it is automatically truncated.

Specify all field names in the NAME() attribute without a prefix.

- ❖ dBase IV additionally supports the use of the Xbase FOR statement in expressions to define keys. The expressions supported in the FOR condition must be a simple condition of the form:

expression comparison\_op expression

*comparison\_op* may be <, <=, =<, <>, =, =>, >= or >.

The expression may refer to multiple fields in the record and contain xBase functions. Square brackets must enclose the expression. The currently supported functions appear below. If the driver encounters an unsupported Xbase function in a preexisting file, it posts error 76 'Invalid Index String' when the file is opened for keys and static indexes.

String expressions may use the '+' operator to concatenate multiple string arguments. Numeric expressions use the '+' or '-' operators with their conventional meanings. The maximum length of a dBase IV expression is 250 characters.

### Supported xBase Key Definition Functions

|                 |                                                                                                                                                                                                                                              |
|-----------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ALLTRIM(string) | Removes leading and trailing spaces.                                                                                                                                                                                                         |
| CTOD(string)    | Converts a string key to a date. The <i>string</i> must be in the format mm/dd/yy; the result takes the form 'yyyymmdd'. The yyyy element of the date defaults to the twentieth century. An invalid date results in a key containing blanks. |
| DELETED()       | Returns TRUE if the record is deleted.                                                                                                                                                                                                       |
| DTOC(date)      | Converts a date key to string format 'mm/dd/yy.'                                                                                                                                                                                             |

|                                          |                                                                                                                                                                                 |
|------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| DTOS(date)                               | Converts a date key to string format 'yyyymmdd.'                                                                                                                                |
| FIXED(float)                             | Converts a float key to a numeric.                                                                                                                                              |
| FLOAT(numeric)                           | Converts a numeric key to a float.                                                                                                                                              |
| IIF(bool, val1, val2)                    | Returns val1 if the first parameter is TRUE, otherwise returns val2.                                                                                                            |
| LEFT(string, n)                          | Returns the leftmost <i>n</i> characters of the string key as a string of length <i>n</i> .                                                                                     |
| LOWER(string)                            | Converts a string key to lower case.                                                                                                                                            |
| LTRIM(string)                            | Removes spaces from the left of a string.                                                                                                                                       |
| RECNO()                                  | Returns the current record number.                                                                                                                                              |
| RIGHT(string, n)                         | Returns the rightmost <i>n</i> characters of the string key as a string of length <i>n</i> .                                                                                    |
| RTRIM(string)                            | Removes spaces from the right of a string.                                                                                                                                      |
| STR(numeric [,length[, decimal places]]) | converts a numeric to a string. The length of the string and the number of decimal places are optional. The default string length is 10, and the number of decimal places is 0. |
| SUBSTR(string, offset, n)                | Returns a substring of the <i>string</i> key starting at <i>offset</i> and of <i>n</i> characters in length.                                                                    |
| TRIM(string)                             | Removes spaces from the right of a string (identical to RTRIM).                                                                                                                 |
| UPPER(string)                            | Converts a string key to upper case.                                                                                                                                            |
| VAL(string)                              | Converts a string key to a numeric.                                                                                                                                             |

Copies the UNNAMED (the only table in the file) table from CUSTOMER.TPS (which has no password) to the CUSTOMER table in ORDERS.TPS which has the password acme96.

## Configurable **ERROR** Messages

All of TopSpeed's database drivers post error conditions and messages that can be accessed with the `ERRORCODE()`, `ERROR()`, `FILEERRORCODE()` and `FILEERROR()` procedures (see the *Language Reference*). The drivers post the codes immediately after each I/O (`OPEN`, `NEXT`, `GET`, `ADD`, `DELETE`, etc.) operation.

**Tip:** The `ERRORFILE()` procedure returns the file or table that produced the error.

See *Run Time Errors* in the *Language Reference* for information on the `ERRORCODE()` values and their corresponding `ERROR()` message text. The `ERROR()` text is configurable using the environment file (`CLAMSG`) and the `LOCALE` procedure. See *Internationalization*, `CLAMSG`, and `LOCALE` in the *Language Reference* for more information.

In addition, dBaseIV driver `FILEERROR()` message text is also configurable using the environment file (`CLAMSG`) and the `LOCALE` procedure. However, the `CLAMSG` value required to change the message text is equal to the `FILEERRORCODE()` value + 4000. Following is a list of the `FILEERROR()` text, the `FILEERRORCODE()` values, and the `CLAMSG` values required to change the `FILEERROR()` text:

| <code>FILEERRORCODE</code> | <code>CLAMSG</code> | <code>FILEERROR</code>            |
|----------------------------|---------------------|-----------------------------------|
| 1                          | 4001                | file write failure                |
| 2                          | 4002                | file read failure                 |
| 3                          | 4003                | memory allocation error           |
| 4                          | 4004                | file pointer reposition failed    |
| 5                          | 4005                | file not found                    |
| 6                          | 4006                | file corrupted                    |
| 7                          | 4007                | bad user specified key expression |
| 8                          | 4008                | no handles available              |
| 9                          | 4009                | no index pages loaded             |
| 10                         | 4010                | index page was not loaded         |
| 11                         | 4011                | file close failure                |
| 12                         | 4012                | invalid command                   |
| 13                         | 4013                | invalid handle number             |
| 14                         | 4014                | invalid filename                  |
| 15                         | 4015                | invalid date                      |
| 16                         | 4016                | invalid time                      |
| 17                         | 4017                | file not in memo format           |
| 18                         | 4018                | invalid dBaseIV version           |
| 19                         | 4019                | file header length error          |
| 20                         | 4020                | last file change date in error    |
| 21                         | 4021                | parameter address NULL            |
| 22                         | 4022                | invalid key type                  |
| 23                         | 4023                | invalid key length                |

## FILEERRORCODE CLAMSG

## FILEERROR

|    |      |                                          |
|----|------|------------------------------------------|
| 24 | 4024 | item length incorrect                    |
| 24 | 4025 | invalid root page                        |
| 24 | 4026 | bad maximum number of keys per page      |
| 24 | 4027 | invalid number of fields                 |
| 24 | 4028 | field name invalid                       |
| 24 | 4029 | bad field length                         |
| 24 | 4030 | decimal places parameter invalid         |
| 24 | 4031 | invalid field type                       |
| 24 | 4032 | invalid record length                    |
| 24 | 4033 | bad data                                 |
| 24 | 4034 | memo soft line length invalid            |
| 24 | 4035 | MDX flag in DBF file invalid             |
| 24 | 4036 | file open for reading only               |
| 24 | 4037 | file locking violation                   |
| 24 | 4038 | sharing buffer overflow                  |
| 24 | 4039 | path not found                           |
| 24 | 4040 | access to file denied                    |
| 24 | 4041 | invalid access code                      |
| 24 | 4042 | file must be locked first                |
| 24 | 4043 | diskette changed                         |
| 24 | 4044 | bad minimum number of keys per page      |
| 24 | 4045 | some files remain open                   |
| 24 | 4046 | could not open the file                  |
| 24 | 4047 | flush to disk failure                    |
| 24 | 4048 | invalid tag handle                       |
| 24 | 4049 | invalid page size in blocks              |
| 24 | 4050 | invalid tag name                         |
| 24 | 4051 | invalid page offset adder                |
| 24 | 4052 | invalid max number of tag table elements |
| 24 | 4053 | bad tag table element length             |
| 24 | 4054 | invalid tag count                        |
| 24 | 4055 | unknown key format switches              |
| 24 | 4056 | unknown switch error                     |
| 24 | 4057 | tag in use                               |
| 24 | 4058 | Windows GlobalLock error                 |
| 24 | 4059 | Windows Task lookup Failed               |
| 24 | 4060 | internal lock buffer overflow            |
| 24 | 4061 | internal lock buffer underflow           |

# 31 - DOS DATABASE DRIVER

## Specifications

The DOS file driver reads and writes any binary, byte-addressable files. Neither fields nor records are delimited. When reading a record, the driver reads the number of bytes defined in the file's RECORD structure, unless a length parameter is specified in the GET statement.

The DOS driver supports the length parameter for the ADD, APPEND, GET, and PUT statements; this allows for variable length records in a DOS file.

The POINTER function returns the relative byte position within the file of the beginning of the last record accessed by an ADD, APPEND, GET, or NEXT statement.

This file driver performs forward sequential processing *only*. No key or transaction processing functions are supported, and the PREVIOUS statement is not supported.

**Tip:** Due to its limitations, the main function of this driver is as a disk editor for binary files.

## Files

---

|                    |                                       |
|--------------------|---------------------------------------|
| <b>C4DOSL.LIB</b>  | Windows Static Link Library (16-bit)  |
| <b>C4DOSXL.LIB</b> | Windows Static Link Library (32-bit)  |
| <b>C4DOS.LIB</b>   | Windows Export Library (16-bit)       |
| <b>C4DOSX.LIB</b>  | Windows Export Library (32-bit)       |
| <b>C4DOS.DLL</b>   | Windows Dynamic Link Library (16-bit) |
| <b>C4DOSX.DLL</b>  | Windows Dynamic Link Library (32-bit) |

## Data Types

---

|         |          |
|---------|----------|
| BYTE    | DECIMAL  |
| SHORT   | PDECIMAL |
| USHORT  | STRING   |
| LONG    | CSTRING  |
| ULONG   | PSTRING  |
| SREAL   | DATE     |
| REAL    | TIME     |
| BFLOAT4 | GROUP    |
| BFLOAT4 |          |

## File Specifications/Maximums

---

|                        |   |                            |
|------------------------|---|----------------------------|
| File Size              | : | 4,294,967,295              |
| Records per File       | : | 4,294,967,295              |
| Record Size            | : | 64K                        |
| Field Size             | : | 64K                        |
| Fields per Record      | : | 64K                        |
| Keys/Indexes per File: |   | n/a                        |
| Key Size               | : | n/a                        |
| Memo fields per File:  |   | n/a                        |
| Memo Field Size        | : | n/a                        |
| Open Data Files        | : | Operating system dependent |

## Driver Strings

There are switches or “driver strings” you can set to control the way your application creates, reads, and writes files with a specific driver. Driver strings are simply messages or parameters that are sent to the file driver at run-time to control its behavior. See *Common Driver Features—Driver Strings* for an overview of these runtime Database Driver switches and parameters.

**Note:** Some driver strings have no effect after the file is open, so no SEND function syntax is listed for those strings. However, the SEND function syntax to return the value of the switch is listed for all driver strings.

The DOS Driver supports the following Driver Strings:

### FILEBUFFERS

```
DRIVER('DOS', '/FILEBUFFERS = n')
[Buffers" =] SEND(file, 'FILEBUFFERS [= n]')
```

Sets the size of the buffer used to read and write to the file, where the buffer size is (n \* 512 bytes). Use the /FILEBUFFERS driver string to increase the buffer size if access is slow. Maximum buffer size is 65,504 in 16-bit and 4,294,967,264 in 32-bit. SEND returns the size of the buffer in bytes.

**Tip:** The default buffer size for files opened denying write access to other users is the larger of 1024 or (2 \* record size), and the larger of 512 or record size for all other open modes.

### QUICKSCAN

```
DRIVER('DOS', '/QUICKSCAN = on | off')
[QScan" =] SEND(file, 'QUICKSCAN [= on | off]')
```

Specifies buffered access behavior. The ASCII driver reads a buffer at a time (not a record), allowing faster access. In a multi-user environment these buffers are not 100% trustworthy for subsequent access, because another user may change the file between accesses. As a safeguard, the driver rereads the buffers before each record access. To *disable* the reread, set QUICKSCAN to ON. The default is ON for files opened denying write access to other users, and OFF for all other open modes. SEND returns the Quickscan setting (ON or OFF) in the form of a STRING(3).

## Supported Attributes and Procedures

### File Attributes Supported

|                                                          |                |
|----------------------------------------------------------|----------------|
| CREATE .....                                             | Y              |
| DRIVER( <i>filetype</i> [, <i>driver string</i> ]) ..... | Y              |
| NAME .....                                               | Y              |
| ENCRYPT .....                                            | N              |
| OWNER( <i>password</i> ) .....                           | N              |
| RECLAIM .....                                            | N              |
| PRE( <i>prefix</i> ) .....                               | Y              |
| BINDABLE .....                                           | Y              |
| THREAD .....                                             | Y <sup>4</sup> |
| EXTERNAL( <i>member</i> ) .....                          | Y              |
| DLL( <i>flag</i> ) .....                                 | Y              |
| OEM .....                                                | Y              |

### File Structures Supported

|              |   |
|--------------|---|
| INDEX .....  | N |
| KEY .....    | N |
| MEMO .....   | N |
| BLOB .....   | N |
| RECORD ..... | Y |

### Index, Key, Memo Attributes Supported

|                             |   |
|-----------------------------|---|
| BINARY .....                | N |
| DUP .....                   | N |
| NOCASE .....                | N |
| OPT .....                   | N |
| PRIMARY .....               | N |
| NAME .....                  | N |
| Ascending Components .....  | N |
| Descending Components ..... | N |
| Mixed Components .....      | N |

### Field Attributes Supported

|            |   |
|------------|---|
| DIM .....  | Y |
| OVER ..... | Y |
| NAME ..... | Y |

### File Procedures Supported

|                                                 |   |
|-------------------------------------------------|---|
| BOF( <i>file</i> ) .....                        | N |
| BUFFER( <i>file</i> ) .....                     | N |
| BUILD( <i>file</i> ) .....                      | N |
| BUILD( <i>key</i> ) .....                       | N |
| BUILD( <i>index</i> ) .....                     | N |
| BUILD( <i>index, components</i> ) .....         | N |
| BUILD( <i>index, components, filter</i> ) ..... | N |
| BYTES( <i>file</i> ) .....                      | Y |
| CLOSE( <i>file</i> ) .....                      | Y |
| COPY( <i>file, new file</i> ) .....             | Y |
| CREATE( <i>file</i> ) .....                     | Y |
| DUPLICATE( <i>file</i> ) .....                  | N |
| DUPLICATE( <i>key</i> ) .....                   | N |
| EMPTY( <i>file</i> ) .....                      | Y |
| EOF( <i>file</i> ) .....                        | Y |
| FLUSH( <i>file</i> ) .....                      | N |
| LOCK( <i>file</i> ) .....                       | Y |
| NAME( <i>label</i> ) .....                      | Y |
| OPEN( <i>file, access mode</i> ) .....          | Y |

|                                         |                |
|-----------------------------------------|----------------|
| PACK( <i>file</i> ) .....               | N              |
| POINTER( <i>file</i> ) .....            | Y <sup>2</sup> |
| POINTER( <i>key</i> ) .....             | N              |
| POSITION( <i>file</i> ) .....           | Y <sup>3</sup> |
| POSITION( <i>key</i> ) .....            | N              |
| RECORDS( <i>file</i> ) .....            | Y              |
| RECORDS( <i>key</i> ) .....             | N              |
| REMOVE( <i>file</i> ) .....             | Y              |
| RENAME( <i>file, new file</i> ) .....   | Y              |
| SEND( <i>file, message</i> ) .....      | Y              |
| SHARE( <i>file, access mode</i> ) ..... | Y              |
| STATUS( <i>file</i> ) .....             | Y              |
| STREAM( <i>file</i> ) .....             | N              |
| UNLOCK( <i>file</i> ) .....             | Y              |

### Record Access Supported

|                                               |                |
|-----------------------------------------------|----------------|
| ADD( <i>file</i> ) .....                      | Y              |
| ADD( <i>file, length</i> ) .....              | Y              |
| APPEND( <i>file</i> ) .....                   | Y              |
| APPEND( <i>file, length</i> ) .....           | Y              |
| DELETE( <i>file</i> ) .....                   | N              |
| GET( <i>file, key</i> ) .....                 | N              |
| GET( <i>file, filepointer</i> ) .....         | Y              |
| GET( <i>file, filepointer, length</i> ) ..... | Y              |
| GET( <i>key, keypointer</i> ) .....           | N              |
| HOLD( <i>file</i> ) .....                     | N              |
| NEXT( <i>file</i> ) .....                     | Y              |
| NOMEMO( <i>file</i> ) .....                   | N              |
| PREVIOUS( <i>file</i> ) .....                 | Y              |
| PUT( <i>file</i> ) .....                      | Y              |
| PUT( <i>file, filepointer</i> ) .....         | Y <sup>1</sup> |
| PUT( <i>file, filepointer, length</i> ) ..... | Y <sup>1</sup> |
| RELEASE( <i>file</i> ) .....                  | N              |
| REGET( <i>file, string</i> ) .....            | Y              |
| REGET( <i>key, string</i> ) .....             | N              |
| RESET( <i>file, string</i> ) .....            | Y              |
| RESET( <i>key, string</i> ) .....             | N              |
| SET( <i>file</i> ) .....                      | Y              |
| SET( <i>file, key</i> ) .....                 | N              |
| SET( <i>file, filepointer</i> ) .....         | Y              |
| SET( <i>key</i> ) .....                       | N              |
| SET( <i>key, key</i> ) .....                  | N              |
| SET( <i>key, keypointer</i> ) .....           | N              |
| SET( <i>key, key, filepointer</i> ) .....     | N              |
| SKIP( <i>file, count</i> ) .....              | N              |
| WATCH( <i>file</i> ) .....                    | N              |

### Transaction Processing Supported

|                                                 |   |
|-------------------------------------------------|---|
| LOGOUT( <i>timeout, file, ..., file</i> ) ..... | N |
| COMMIT .....                                    | N |
| ROLLBACK .....                                  | N |

### Null Data Processing Supported

|                                  |   |
|----------------------------------|---|
| NULL( <i>field</i> ) .....       | N |
| SETNULL( <i>field</i> ) .....    | N |
| SETNONNULL( <i>field</i> ) ..... | N |

## Notes

---

- 1 When using PUT() with this driver you should take care to PUT back the same number of characters that were read. If you PUT back more characters than were read, then the “extra” characters will overwrite the first part of the subsequent record. If you PUT back fewer characters than were read, then only the first part of the retrieved record is overwritten, while the last part of the retrieved record remains as it was prior to the PUT().
- 2 POINTER() returns the relative byte position within the file.
- 3 POSITION(file) returns a STRING(4).
- 4 THREADED files consume additional file handles for each thread that accesses the file.



# 32 - FoxPro / FoxBase Database Driver

## Specifications

The FoxPro file driver is compatible with FoxPro and FoxBase. The default data file extension is \*.DBF.

The default index file extension is \*.IDX. The default Memo file extension is .FBT. FoxPro also supports multiple index files, whose default extension is \*.CDX. The miscellaneous section describes the procedures for using the .CDX files.

## Files

---

|                   |                                       |
|-------------------|---------------------------------------|
| <b>C5FOX.LIB</b>  | Windows Static Link Library (16-bit)  |
| <b>C5FOXX.LIB</b> | Windows Static Link Library (32-bit)  |
| <b>C5FOX.LIB</b>  | Windows Export Library (16-bit)       |
| <b>C5FOXX.LIB</b> | Windows Export Library (32-bit)       |
| <b>C5FOX.DLL</b>  | Windows Dynamic Link Library (16-bit) |
| <b>C5FOXX.DLL</b> | Windows Dynamic Link Library (32-bit) |

**Tip:** The FoxPro index file format is the backbone of its vaunted “Rushmore” technology. The old saying “There’s no free lunch,” however, applies. Adding and appending records to a large database is a slower process than in other xBase formats, due to the time required to update the index file.

## Data Types

---

The xBase file format stores all data as ASCII strings. You may either specify STRING types with declared pictures for each field, or specify native Clarion types, which the driver converts automatically.

| <u>FoxPro data type</u> | <u>Clarion data type</u> | <u>STRING w/ picture</u> |
|-------------------------|--------------------------|--------------------------|
| Date                    | DATE                     | STRING(@D12)             |
| *Numeric                | REAL                     | STRING(@N-_p.d)          |
| *Logical                | BYTE                     | STRING(1)                |
| Character               | STRING                   | STRING                   |
| *Memo                   | MEMO                     | MEMO                     |

If your application reads and writes to existing files, a pictured STRING will suffice. However, if your application *creates* a FoxPro or FoxBase file, you may require additional information for these FoxPro and FoxBase types:

- ❖ To create a numeric field in the Data Dictionary, choose the REAL data type. In the External Name field on the Attributes tab, specify '*NumericFieldName*=N(*Precision*,*DecimalPlaces*)' where *NumericFieldName* is the name of the field, *Precision* is the precision of the field and *DecimalPlaces* is the number of decimal places. With a REAL data type, you cannot access the Character or Places fields in the Field definition, you must specify those attributes with an expression in the External Name Field on the Attributes tab.

For example, if you want to create a field called Number with nine significant digits and two decimal places, enter 'Number=N(9,2)' in the External Name field on the Attributes tab of the Field properties in the Data Dictionary.

If you're hand coding a native Clarion data type, add the NAME attribute using the same syntax.

If you're hand coding a STRING with picture, STRING(@N-\_9.2), NAME('Number'), where *Number* is the field name.

- ❖ To create a logical field, using the data dictionary, choose the BYTE data type. There are no special steps; however, see the miscellaneous section for tips on reading the data from the field.

If you're hand coding a STRING with picture, add the NAME attribute: STRING(1), NAME('LogFld = L').

- ❖ To create a date field, using the data dictionary, choose the DATE data type, rather than LONG, which you usually use for the TopSpeed or Clarion file formats.
- ❖ MEMO field declarations require the a pointer field in the file's record structure. Declare the pointer field as a STRING(10) or a LONG. This field will be stored in the .DBF file containing the offset of the memo in the .DBT file. The MEMO declaration must have a NAME() attribute naming the pointer field. An example file declaration follows:

```
File FILE, DRIVER('FoxPro')
Memo1 MEMO(200),NAME('Notes')
Memo2 MEMO(200),NAME('Text')
Rec RECORD
Mem1Ptr LONG,NAME('Notes')
Mem2Ptr STRING(10),NAME('Text')
END
END
```

## File Specifications/Maximums

---

|                        |                                      |
|------------------------|--------------------------------------|
| File Size:             | 2,000,000,000 bytes                  |
| Records per File:      | 1,000,000,000 bytes                  |
| Record Size:           | 4,000 bytes                          |
| Field Size             |                                      |
| Character:             | 254 bytes                            |
| Date:                  | 8 bytes                              |
| Logical:               | 1 byte                               |
| Numeric:               | 20 bytes including decimal point     |
| Float:                 | 20 bytes including decimal point     |
| Memo:                  | 65,520 bytes (see note)              |
| Fields per Record:     | 512                                  |
| Keys/Indexes per File: | No Limit                             |
| Key Sizes              |                                      |
| Character:             | 100 bytes (.IDX)<br>254 bytes (.CDX) |
| Numeric, Date:         | 8 bytes                              |
| Memo fields per File:  | Dependent on available memory        |
| Open Files:            | Operating system dependent           |

## Driver Strings

There are switches or “driver strings” you can set to control the way your application creates, reads, and writes files with a specific driver. Driver strings are simply messages or parameters that are sent to the file driver at run-time to control its behavior. See *Common Driver Features—Driver Strings* for an overview of these runtime Database Driver switches and parameters.

**Note:** Some driver strings have no effect after the file is open, so no SEND function syntax is listed for those strings. However, the SEND function syntax to return the value of the switch is listed for all driver strings.

The FoxPro Driver supports the following Driver Strings:

## BUFFERS

```
DRIVER('FOXPRO', '/BUFFERS = n')
[Status" =] SEND(file, 'BUFFERS [= n]')
```

Sets the size of the buffer used to read and write to the file, where the buffer size is (n \* 512 bytes). Use the /BUFFERS driver string to increase the buffer size if access is slow. Maximum buffer size is 65,504 in 16-bit and 4,294,967,264 in 32-bit. SEND returns the size of the buffer in bytes.

**Tip:** The default is three buffers of 1024 bytes each. Increasing the number of buffers will not increase performance when a file is shared by multiple users.

## RECOVER

```
DRIVER('FOXPRO', '/RECOVER')
[Status" =] SEND(file, 'RECOVER')
```

Equivalent to the Xbase RECALL command, which recovers records marked for deletion. When using the FoxPro driver, the DELETE statement flags a record as “inactive.” The driver does not remove the record until the PACK command is executed.

RECOVER is evaluated each time you open the file if you add the driver string to the data dictionary. When the driver recovers the records previously marked for deletion, you must manually rebuild keys and indexes with the BUILD statement.

## IGNORESTATUS

---

```
DRIVER('FOXPRO', '/IGNORESTATUS = on | off ')
[Status" =] SEND(file, 'IGNORESTATUS [on | off]')
```

When set *on*, the driver does *not* skip deleted records when accessing the file with GET, NEXT, and PREVIOUS in file order. It also enables a PUT on a deleted or held record. IGNORESTATUS requires opening the file in exclusive mode. SEND returns the IGNORESTATUS setting (ON or OFF) in the form of a STRING(3).

## DELETED

---

```
[Status" =] SEND(file, 'DELETED')
```

For use only with the SEND command, when IGNORESTATUS is on. Returns the status of the current record. If deleted, the return string is "ON" and if not, "OFF."

## Supported Attributes and Procedures

### File Attributes Supported

|                                                          |                 |
|----------------------------------------------------------|-----------------|
| CREATE .....                                             | Y               |
| DRIVER( <i>filetype</i> [, <i>driver string</i> ]) ..... | Y               |
| NAME .....                                               | Y               |
| ENCRYPT .....                                            | N               |
| OWNER( <i>password</i> ) .....                           | N               |
| RECLAIM .....                                            | N <sup>1</sup>  |
| PRE( <i>prefix</i> ) .....                               | Y               |
| BINDABLE .....                                           | Y               |
| THREAD .....                                             | Y <sup>13</sup> |
| EXTERNAL( <i>member</i> ) .....                          | Y               |
| DLL( <i>flag</i> ) .....                                 | Y               |
| OEM .....                                                | N <sup>2</sup>  |

### File Structures Supported

|              |   |
|--------------|---|
| INDEX .....  | Y |
| KEY .....    | Y |
| MEMO .....   | Y |
| BLOB .....   | N |
| RECORD ..... | Y |

### Index, Key, Memo Attributes Supported

|                             |                 |
|-----------------------------|-----------------|
| BINARY .....                | N <sup>14</sup> |
| DUP .....                   | Y <sup>3</sup>  |
| NOCASE .....                | Y               |
| OPT .....                   | N               |
| PRIMARY .....               | Y               |
| NAME .....                  | Y               |
| Ascending Components .....  | Y               |
| Descending Components ..... | Y               |
| Mixed Components .....      | N               |

### Field Attributes Supported

|            |   |
|------------|---|
| DIM .....  | N |
| OVER ..... | Y |
| NAME ..... | Y |

### File Procedures Supported

|                                                 |                |
|-------------------------------------------------|----------------|
| BOF( <i>file</i> ) .....                        | Y <sup>9</sup> |
| BUFFER( <i>file</i> ) .....                     | N              |
| BUILD( <i>file</i> ) .....                      | Y              |
| BUILD( <i>key</i> ) .....                       | Y              |
| BUILD( <i>index</i> ) .....                     | Y              |
| BUILD( <i>index, components</i> ) .....         | Y <sup>4</sup> |
| BUILD( <i>index, components, filter</i> ) ..... | N              |
| BYTES( <i>file</i> ) .....                      | N              |
| CLOSE( <i>file</i> ) .....                      | Y              |
| COPY( <i>file, new file</i> ) .....             | Y <sup>5</sup> |
| CREATE( <i>file</i> ) .....                     | Y              |
| DUPLICATE( <i>file</i> ) .....                  | Y              |
| DUPLICATE( <i>key</i> ) .....                   | Y              |
| EMPTY( <i>file</i> ) .....                      | Y              |
| EOF( <i>file</i> ) .....                        | Y <sup>9</sup> |
| FLUSH( <i>file</i> ) .....                      | Y              |
| LOCK( <i>file</i> ) .....                       | N              |
| NAME( <i>label</i> ) .....                      | Y              |
| OPEN( <i>file, access mode</i> ) .....          | Y <sup>6</sup> |

|                                         |                 |
|-----------------------------------------|-----------------|
| PACK( <i>file</i> ) .....               | Y               |
| POINTER( <i>file</i> ) .....            | Y <sup>10</sup> |
| POINTER( <i>key</i> ) .....             | Y <sup>10</sup> |
| POSITION( <i>file</i> ) .....           | Y <sup>11</sup> |
| POSITION( <i>key</i> ) .....            | Y <sup>11</sup> |
| RECORDS( <i>file</i> ) .....            | Y <sup>12</sup> |
| RECORDS( <i>key</i> ) .....             | Y <sup>12</sup> |
| REMOVE( <i>file</i> ) .....             | Y               |
| RENAME( <i>file, new file</i> ) .....   | Y <sup>5</sup>  |
| SEND( <i>file, message</i> ) .....      | Y               |
| SHARE( <i>file, access mode</i> ) ..... | Y <sup>6</sup>  |
| STATUS( <i>file</i> ) .....             | Y               |
| STREAM( <i>file</i> ) .....             | Y               |
| UNLOCK( <i>file</i> ) .....             | Y               |

### Record Access Supported

|                                               |                |
|-----------------------------------------------|----------------|
| ADD( <i>file</i> ) .....                      | Y <sup>7</sup> |
| ADD( <i>file, length</i> ) .....              | N              |
| APPEND( <i>file</i> ) .....                   | Y <sup>7</sup> |
| APPEND( <i>file, length</i> ) .....           | N              |
| DELETE( <i>file</i> ) .....                   | Y <sup>1</sup> |
| GET( <i>file, key</i> ) .....                 | Y              |
| GET( <i>file, filepointer</i> ) .....         | Y              |
| GET( <i>file, filepointer, length</i> ) ..... | N              |
| GET( <i>key, keypointer</i> ) .....           | Y              |
| HOLD( <i>file</i> ) .....                     | Y <sup>8</sup> |
| NEXT( <i>file</i> ) .....                     | Y              |
| NOMEMO( <i>file</i> ) .....                   | Y              |
| PREVIOUS( <i>file</i> ) .....                 | Y              |
| PUT( <i>file</i> ) .....                      | Y              |
| PUT( <i>file, filepointer</i> ) .....         | Y              |
| PUT( <i>file, filepointer, length</i> ) ..... | N              |
| RELEASE( <i>file</i> ) .....                  | Y              |
| REGET( <i>file, string</i> ) .....            | Y              |
| REGET( <i>key, string</i> ) .....             | Y              |
| RESET( <i>file, string</i> ) .....            | Y              |
| RESET( <i>key, string</i> ) .....             | Y              |
| SET( <i>file</i> ) .....                      | Y              |
| SET( <i>file, key</i> ) .....                 | Y              |
| SET( <i>file, filepointer</i> ) .....         | Y              |
| SET( <i>key</i> ) .....                       | Y              |
| SET( <i>key, key</i> ) .....                  | Y              |
| SET( <i>key, keypointer</i> ) .....           | Y              |
| SET( <i>key, key, filepointer</i> ) .....     | Y              |
| SKIP( <i>file, count</i> ) .....              | Y              |
| WATCH( <i>file</i> ) .....                    | Y              |

### Transaction Processing Supported<sup>15</sup>

|                                                 |   |
|-------------------------------------------------|---|
| LOGOUT( <i>timeout, file, ..., file</i> ) ..... | N |
| COMMIT .....                                    | N |
| ROLLBACK .....                                  | N |

### Null Data Processing Supported

|                                  |   |
|----------------------------------|---|
| NULL( <i>field</i> ) .....       | N |
| SETNULL( <i>field</i> ) .....    | N |
| SETNONNULL( <i>field</i> ) ..... | N |

## Notes

---

- 1 When the driver deletes a record from a FoxPro database, the record is not physically removed, instead the driver marks it inactive. Memo fields are not physically removed from the memo file, however they cannot be retrieved if they refer to an inactive record. Key values *are* removed from the index files. To remove records and memo fields permanently, execute a PACK(file).

**Tip:** To those programmers familiar with FoxPro, this driver processes deleted records consistent with the way FoxPro processes them after the SET DELETED ON command is issued. Records marked for deletion are ignored from processing by executable code statements, but remain in the data file.

- 2 If you need to access FoxPro data with alternate characters stored using a non-English version of FoxPro, then you should use ODBC. However, if you do not have any string based kesys, you can use the FoxPro driver and call the ConvertOEMToANSI and ConvertANSIToOEM after retrieving and before updating a record.
- 3 In FoxPro it is legal to enter multiple records with duplicates of the unique key components. However, only the first of these records is indexed. So processing in key order only shows this first record. If you delete a record, then enter a new record with the same key value, the key file continues to point at the deleted record rather than the new record. In this situation, the FoxPro file driver changes the key file to point at the active record rather than the deleted record. This means that if you use a FoxPro program to delete a unique record, then insert a duplicate of this record, the new record is invisible when processing in key order until a pack is done. If you do the same process in a Clarion program, the new record is visible when processing in key order.
- 4 When building dynamic indexes, the *components* may take one of two forms:  

```
BUILD(DynNdx, '+Pre:FLD1, -Pre:FLD2')
```

This form specifies the names of the fields on which to build the index. The field names must appear as specified in the fields' NAME() attribute if supplied, or must be the label name. A prefix may be used for compatibility with Clarion conventions but is ignored.

```
BUILD(DynNdx, 'T[Expression]')
```

This form specifies the type and Expression used to build an index—see *Miscellaneous—Key Definition* for more information.
- 5 These commands copy data and memo files using *newfile*, which may specify a new file name or directory. Key or index files are copied if the *newfile* is a subdirectory specification. To copy an index file to a new

file, use a special form of the copy command:

```
COPY(file, '<index>|<newfile>')
```

This returns *File Not Found* if an invalid index is passed. The COPY command assumes a default extension of .IDX for both the source and the target file names if none is specified. If you require a file name without an extension, terminate the name with a period. Given the file structure:

```
CLar2 FILE, CREATE, DRIVER('FoxPro'), PRE(CL2)
NumKey KEY(+CL2:Num), DUP
StrKey KEY(+CL2:Str1)
StrKey2 KEY(+CL2:Str2)
AMemo MEMO(100), NAME('mem')
Record RECORD
Num STRING(@n-_9.2)
STR1 STRING(2)
STR2 STRING(2)
Mem STRING(10)
. . .
```

The following commands copy this file definition to A:

```
COPY(CLar2, 'A:\CLAR2')
COPY(CLar2, 'StrKey|A:\STRKEY')
COPY(CLar2, 'StrKey2|A:\STRKEY2')
COPY(CLar2, 'NumKey|A:\NUMKEY')
```

After these calls, the following files would exist on drive A: CLAR2.DBF, CLAR2.FPT, STRKEY.IDX, STRKEY2.IDX, and NUMKEY.IDX.

- 6 You do not need SHARE (or VSHARE) in any environment (for example, Novell) that supplies file locking as part of the operating system.
- 7 The ADD statement tests for duplicate keys before modifying the data file or its associated KEY files. Consequently it is slower than APPEND which performs no checks and does not update KEYS. When adding large amounts of data to a database use APPEND...BUILD in preference to ADD.
- 8 FoxPro performs record locking by locking the entire record within the data file. This prevents read access to other processes. Therefore we recommend minimizing the amount of time for which a record is held.
- 9 Although the driver supports these functions, we do not recommend their use. They must physically access the files and they are slow. Instead, test the value returned by ERRORCODE() after each sequential access. NEXT or PREVIOUS post Error 33 (Record Not Available) if an attempt is made to access a record beyond the end or beginning of the file.
- 10 There is no distinction between file pointers and key pointers; they both return the record number for the given record.
- 11 POSITION(file) returns a STRING(12). POSITION(key) returns a

STRING the size of the key fields + 4 bytes.

- 12 Under FoxPro, the RECORDS() function reports the same number of records for the data file and its keys and indexes. Usually there will be no difference in the number of records *unless* the INDEX is out of date. Because the DELETE statement does not physically remove records, *the number of records reported by the RECORDS() function includes inactive records*. Exercise care when using this function. The field names must appear as specified in the fields' NAME() attribute if supplied, or must be the label name. A prefix may be used for compatibility with the Clarion conventions but is ignored.
- 13 THREADED files consume additional file handles for each thread that accesses the file.
- 14 OEM conversion is not applied to BINARY MEMOs. The driver assumes BINARY MEMOs are zero padded; otherwise, space padded.
- 15 You should clear the **Enclose RI code in transaction frame** box on the **File Control** tab of the **Global Properties** dialog when using this driver with template generated applications.

## Other

### Boolean Evaluation

- ❖ FoxPro and FoxBase allow a logical field to accept one of 11 possible values (0,1,y,Y,n,N,t,T,f,F or a space character). The space character is neither true nor false. When using a logical field from a preexisting database in a logical expression, account for all these possibilities. Remember that when a STRING field is used as an expression, it is true if it contains any data and false if it is equal to zero or blank. Therefore, to evaluate a Logical field's truth, the expression should be true if the field contains any of the "true" characters (1,T,t,Y, or y). For example, if a Logical field were used to specify a product as taxable or nontaxable, the expression to evaluate its truth would be:

*(If Condition):*

```
Taxable='1' OR Taxable='T' OR Taxable='t' OR Taxable='Y' OR Taxable='y'
```

### Large MEMOs

- ❖ Clarion supports MEMO fields up to a maximum of 64K. If you have an existing file which includes a memo greater than 64K, you can use the file but not modify the large MEMOs.
- ❖ You can determine when your application encounters a large MEMO by detecting when the memo pointer variable is non-blank, but the memo appears to be blank. Error 47 (Bad Record Declaration) is posted, and any modification to the MEMO field is ignored.

## Long Field Names

- ❖ FoxPro and FoxBase support a maximum of 10 characters in a field name. If you require more, use an External Name with 10 characters or less.

## Key Definition

- ❖ FoxPro and FoxBase support the use of expressions to define keys. Within the Dictionary Editor, you can place the expression in the external name field in the *Key Properties* dialog. The general format of the external name is :

```
'FileName=T[Expression]'
```

Where `FileName` represents the name of the index file (which can contain a path and file extension), and `T` represents the type of the index. Valid types are: `C` = character, `D` = date, and `N` = numeric. If the type is `D` or `N` then `Expression` can name only one field.

- ❖ Multiple-index (.CDX) files require the `NAME()` attribute on a `KEY` or `INDEX` to specify the storage type of the key and any expression used to generate the key values. The general format of the `NAME()` attribute on a `KEY` or `INDEX` is:

```
NAME('TagName|FileName[PageSize]=T[Expression],COMPRESSED')
```

The following are the parameters for the `NAME()` attribute:

|                         |                                                                                                                                                                                                                                                                                                                                                                                                  |
|-------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>TagName</code>    | Names an index tag within a multiple index file. If the <code>TagName</code> is omitted the driver creates an .IDX file with the name specified in <code>FileName</code> .                                                                                                                                                                                                                       |
| <code>FileName</code>   | Names the index file, and optionally contains a path and extension.                                                                                                                                                                                                                                                                                                                              |
| <code>PageSize</code>   | May only be specified when creating a .CDX file (if a <code>TagName</code> is specified). It is a number in the range 2-32 specifying the number of 512-byte blocks in each index page. This value is only used when creating the file. If multiple values are specified with declarations for different tags in the same .MDX file, the largest value will be selected. The default value is 2. |
| <code>T</code>          | Specifies the type of the index; legal types are <code>C</code> = character, <code>D</code> = date, <code>N</code> = numeric. If the type is <code>D</code> or <code>N</code> then <code>Expression</code> may name only one field.                                                                                                                                                              |
| <code>Expression</code> | Specifies the expression used to generate the index. The expression may refer to multiple fields, and invoke multiple of xBase functions. The functions currently supported are listed below. Square                                                                                                                                                                                             |

brackets must enclose the expression.

COMPRESSED

When specified, the FoxPro Driver creates a FoxPro 2 compatible compressed .IDX file.

Elements of the NAME() attribute may be omitted from the right. When specifying an Expression, the type and name must also be specified. If the Expression is omitted, the driver determines the Expression from the key fields when the file is created, or from the index file when opened.

If the type is omitted, the driver determines the index type from the first key component when the file is created, or from the index file when opened.

If the NAME() attribute is omitted altogether, the index file name is determined from the key label. The path defaults to the same location as the .DBF.

Tag names are limited to 10 characters in length; if the supplied name is too long it is automatically truncated.

All field names in the NAME() attribute must be specified without a prefix.

- ❖ FoxPro additionally supports the use of the Xbase FOR statement in expressions to define keys. The expressions supported in the FOR condition must be a simple condition of the form:

`expression comparison_op expression`

*comparison\_op* may be one of the following: <, <=, =<, <>, =, =>, >= or >.

The expression may refer to multiple fields in the record, and contain xBase functions. Square brackets must enclose the expression. The currently supported functions appear below. If the driver encounters an unsupported Xbase function in a preexisting file, it posts error 76 'Invalid Index String' when the file is opened for keys and static indexes.

String expressions may use the '+' operator to concatenate multiple string arguments. Numeric expressions use the '+' or '-' operators with their conventional meanings. The maximum length of a FoxPro or FoxBase expression is 250 characters.

## Supported xBase Key Definition Functions

|                                          |                                                                                                                                                                                                                                              |
|------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ALLTRIM(string)                          | Removes leading and trailing spaces.                                                                                                                                                                                                         |
| CTOD(string)                             | Converts a string key to a date. The <i>string</i> must be in the format mm/dd/yy; the result takes the form 'yyyymmdd'. The yyyy element of the date defaults to the twentieth century. An invalid date results in a key containing blanks. |
| DELETED()                                | Returns TRUE if the record is deleted.                                                                                                                                                                                                       |
| DTOC(date)                               | Converts a date key to string format 'mm/dd/yy.'                                                                                                                                                                                             |
| DTOS(date)                               | Converts a date key to string format 'yyyymmdd.'                                                                                                                                                                                             |
| FIXED(float)                             | Converts a float key to a numeric.                                                                                                                                                                                                           |
| FLOAT(numeric)                           | Converts a numeric key to a float.                                                                                                                                                                                                           |
| IIF(bool, val1, val2)                    | Returns val1 if the first parameter is TRUE, otherwise returns val2.                                                                                                                                                                         |
| LEFT(string, n)                          | Returns the leftmost <i>n</i> characters of the string key as a string of length <i>n</i> .                                                                                                                                                  |
| LOWER(string)                            | Converts a string key to lower case.                                                                                                                                                                                                         |
| LTRIM(string)                            | Removes spaces from the left of a string.                                                                                                                                                                                                    |
| RECNO()                                  | Returns the current record number.                                                                                                                                                                                                           |
| RIGHT(string, n)                         | Returns the rightmost <i>n</i> characters of the string key as a string of length <i>n</i> .                                                                                                                                                 |
| RTRIM(string)                            | Removes spaces from the right of a string.                                                                                                                                                                                                   |
| STR(numeric [,length[, decimal places]]) | converts a numeric to a string. The length of the string and the number of decimal places are optional. The default string length is 10, and the number of decimal places is 0.                                                              |
| SUBSTR(string, offset, n)                | Returns a substring of the <i>string</i> key starting at <i>offset</i> and of <i>n</i> characters in length.                                                                                                                                 |
| TRIM(string)                             | Removes spaces from the right of a string (identical to RTRIM).                                                                                                                                                                              |
| UPPER(string)                            | Converts a string key to upper case.                                                                                                                                                                                                         |
| VAL(string)                              | Converts a string key to a numeric.                                                                                                                                                                                                          |

## Configurable **ERROR** Messages

All of TopSpeed's database drivers post error conditions and messages that can be accessed with the `ERRORCODE()`, `ERROR()`, `FILEERRORCODE()` and `FILEERROR()` procedures (see the *Language Reference*). The drivers post the codes immediately after each I/O (`OPEN`, `NEXT`, `GET`, `ADD`, `DELETE`, etc.) operation.

**Tip:** The `ERRORFILE()` procedure returns the file or table that produced the error.

See *Run Time Errors* in the *Language Reference* for information on the `ERRORCODE()` values and their corresponding `ERROR()` message text. The `ERROR()` text is configurable using the environment file (`CLAMSG`) and the `LOCALE` procedure. See *Internationalization*, `CLAMSG`, and `LOCALE` in the *Language Reference* for more information.

In addition, FoxPro driver `FILEERROR()` message text is also configurable using the environment file (`CLAMSG`) and the `LOCALE` procedure. However, the `CLAMSG` value required to change the message text is equal to the `FILEERRORCODE()` value + 4000. Following is a list of the `FILEERROR()` text, the `FILEERRORCODE()` values, and the `CLAMSG` values required to change the `FILEERROR()` text:

| <code>FILEERRORCODE</code> | <code>CLAMSG</code> | <code>FILEERROR</code>            |
|----------------------------|---------------------|-----------------------------------|
| 1                          | 4001                | file write failure                |
| 2                          | 4002                | file read failure                 |
| 3                          | 4003                | memory allocation error           |
| 4                          | 4004                | file pointer reposition failed    |
| 5                          | 4005                | file not found                    |
| 6                          | 4006                | file corrupted                    |
| 7                          | 4007                | bad user specified key expression |
| 8                          | 4008                | no handles available              |
| 9                          | 4009                | no index pages loaded             |
| 10                         | 4010                | index page was not loaded         |
| 11                         | 4011                | file close failure                |
| 12                         | 4012                | invalid command                   |
| 13                         | 4013                | invalid handle number             |
| 14                         | 4014                | invalid filename                  |
| 15                         | 4015                | invalid date                      |
| 16                         | 4016                | invalid time                      |
| 17                         | 4017                | file not in memo format           |
| 18                         | 4018                | invalid FoxPro version            |
| 19                         | 4019                | file header length error          |
| 20                         | 4020                | last file change date in error    |
| 21                         | 4021                | parameter address NULL            |
| 22                         | 4022                | invalid key type                  |
| 23                         | 4023                | invalid key length                |

## FILEERRORCODE CLAMSG

## FILEERROR

|    |      |                                          |
|----|------|------------------------------------------|
| 24 | 4024 | item length incorrect                    |
| 24 | 4025 | invalid root page                        |
| 24 | 4026 | bad maximum number of keys per page      |
| 24 | 4027 | invalid number of fields                 |
| 24 | 4028 | field name invalid                       |
| 24 | 4029 | bad field length                         |
| 24 | 4030 | decimal places parameter invalid         |
| 24 | 4031 | invalid field type                       |
| 24 | 4032 | invalid record length                    |
| 24 | 4033 | bad data                                 |
| 24 | 4034 | memo soft line length invalid            |
| 24 | 4035 | MDX flag in DBF file invalid             |
| 24 | 4036 | file open for reading only               |
| 24 | 4037 | file locking violation                   |
| 24 | 4038 | sharing buffer overflow                  |
| 24 | 4039 | path not found                           |
| 24 | 4040 | access to file denied                    |
| 24 | 4041 | invalid access code                      |
| 24 | 4042 | file must be locked first                |
| 24 | 4043 | diskette changed                         |
| 24 | 4044 | bad minimum number of keys per page      |
| 24 | 4045 | some files remain open                   |
| 24 | 4046 | could not open the file                  |
| 24 | 4047 | flush to disk failure                    |
| 24 | 4048 | invalid tag handle                       |
| 24 | 4049 | invalid page size in blocks              |
| 24 | 4050 | invalid tag name                         |
| 24 | 4051 | invalid page offset adder                |
| 24 | 4052 | invalid max number of tag table elements |
| 24 | 4053 | bad tag table element length             |
| 24 | 4054 | invalid tag count                        |
| 24 | 4055 | unknown key format switches              |
| 24 | 4056 | unknown switch error                     |
| 24 | 4057 | tag in use                               |
| 24 | 4058 | Windows GlobalLock error                 |
| 24 | 4059 | Windows Task lookup Failed               |
| 24 | 4060 | internal lock buffer overflow            |
| 24 | 4061 | internal lock buffer underflow           |

## 33 - TOPSPEED DATABASE DRIVER

### Overview

The TopSpeed Database file system is a high-performance, high-security, proprietary file driver for Clarion development tools. It is *not* file compatible with the Clarion file driver's data.

Data tables, keys, indexes and memos can all be stored together in a single DOS file. The default file extension is \*.TPS. A separate "Transaction Control File" takes the \*.TCF extension.

The TopSpeed driver can optionally store multiple tables in a single file. This lets you open as many data tables, keys, and indexes as necessary using *a single DOS file handle*. This feature may be especially useful when there are a large number of small tables, or when a group of related files are normally accessed together. All keys, indexes, and memos are stored internally.

**Tip:** When multiple tables share a single DOS handle, the first OPEN mode applies to all the tables within the file.

In addition, the TopSpeed file system supports the **BLOB** data type (Binary Large Object), a field which is completely variable-length and may be greater than 64K in size (in both 16-bit and 32-bit applications). A BLOB must be declared before the RECORD structure. Memory for a BLOB is dynamically allocated and de-allocated as necessary. For more information, see *BLOB* in the *Language Reference*.

**Tip:** This driver offers speed, security, and takes up fewer resources on the end user's system.

# Specifications

## Files

---

|                    |                                       |
|--------------------|---------------------------------------|
| <b>C5TPSL.LIB</b>  | Windows Static Link Library (16-bit)  |
| <b>C5TPSXL.LIB</b> | Windows Static Link Library (32-bit)  |
| <b>C5TPS.LIB</b>   | Windows Export Library (16-bit)       |
| <b>C5TPSX.LIB</b>  | Windows Export Library (32-bit)       |
| <b>C5TPS.DLL</b>   | Windows Dynamic Link Library (16-bit) |
| <b>C5TPSX.DLL</b>  | Windows Dynamic Link Library (32-bit) |

## Data Types

---

|        |         |
|--------|---------|
| BYTE   | DECIMAL |
| SHORT  | STRING  |
| USHORT | CSTRING |
| LONG   | PSTRING |
| ULONG  | MEMO    |
| SREAL  | GROUP   |
| REAL   | BLOB    |
| DATE   | TIME    |

## Maximum File Specifications

---

|                            |   |                                                                                |
|----------------------------|---|--------------------------------------------------------------------------------|
| File Size                  | : | Limited only by disk space                                                     |
| Records per File           | : | Unsigned Long (4,294,967,295)                                                  |
| Record Size                | : | 15,000 bytes                                                                   |
| Field Size                 | : | 15,000 bytes                                                                   |
| Fields per Record          | : | 15,000                                                                         |
| Keys/Indexes per File:     |   | 240                                                                            |
| Key Size                   | : | 15,000 bytes                                                                   |
| Memo fields per File:      |   | 255                                                                            |
| Memo Field Size            | : | 64,000 bytes                                                                   |
| BLOB fields per File:      |   | 255                                                                            |
| BLOB Size                  | : | Hardware dependent                                                             |
| Open Data Files            | : | Operating system dependent                                                     |
| Table Name                 | : | 1,000 bytes                                                                    |
| Tables per DOS File        | : | Limited only by the maximum DOS file size—approximately 2 <sup>31</sup> bytes. |
| Concurrent Users per File: |   | 1024                                                                           |

## Driver Strings

There are switches or “driver strings” you can set to control the way your application creates, reads, and writes files with a specific driver. Driver strings are simply messages or parameters that are sent to the file driver at run-time to control its behavior. See *Common Driver Features—Driver Strings* for an overview of these runtime Database Driver switches and parameters.

**Note:** Some driver strings have no effect after the file is open, so no SEND function syntax is listed for those strings. However, the SEND function syntax to return the value of the switch is listed for all driver strings.

The TopSpeed Driver supports the following Driver Strings:

### FLAGS

[ Flags” = ] SEND(file, ‘FLAGS [ = bitmap ]’)

Sets and returns the configuration flags for the *file*. Use the following EQUATEs declared in EQUATES.CLW to control the behavior of the target TopSpeed file:

```
!TopSpeed File Flags
```

```
TPSREADONLY EQUATE(1)
```

For example, the following code makes the file read-only for ODBC access while preserving any other flags:

```
TpsFlags = SEND(MyFile, ‘FLAGS’)
SEND(MyFile, ‘FLAGS =’&BOR(TpsFlags,TPSREADONLY))
```

### FULLBUILD

[ State” = ] SEND(file, ‘FULLBUILD [ = on | off ]’)  
[ State” = ] file{PROP:FULLBUILD} [ = on | off ]’)

The TopSpeed driver has an optimized appending mechanism where you can add large numbers of records to an existing table with the APPEND statement. Issuing a subsequent BUILD updates only the appended key information, making incremental batch updates very fast. This is the default behavior. Use the FULLBUILD driver string to modify this default behavior.

FULLBUILD=ON tells the next BUILD statement to fully rebuild the keys. FULLBUILD=OFF restores the BUILD to its optimized state. Both versions of the SEND command return the current build state as a string ‘ON’ or ‘OFF’. Issue SEND(file,‘FULLBUILD’) to return the current build state without changing it.

## PNM=

**TName" = SEND(file, 'PNM=[starting point]' )**

Returns the next table name in the *file*'s TopSpeed super file, after the specified *starting point*. If there are no table names after the specified *starting point*, SEND returns an empty string. If *starting point* is omitted or contains an empty string, SEND returns the first table name in the file. PNM= is only valid with the SEND command. There are no spaces surrounding the equal sign (=). The target *file* is the label of any of the tables within the TopSpeed super file.

For example, given a TopSpeed file containing the Supp table, the following code displays an alphabetical listing of all the tables in the file:

```
CODE
name = ''
LOOP
 name = SEND(Supp, 'PNM=' & name)
 If name
 MESSAGE(name)
 ELSE
 BREAK
 END
END
END
```

## TCF

**[ TCFLocation" = ] SEND(file, 'TCF [ = filename ]')**

Specifies a transaction control file (TCF) pathname. By default, the transaction control file path is the same as the data file path. The specified TCF identifies all transactions in progress until the program terminates or a SEND(file, 'TCF = filename') executes. In other words, the TCF setting affects all TopSpeed files accessed by the program—it is a global setting. See *Transaction Processing—the TCF File* for more information.

**Tip:** You only need to set the TCF once for any TopSpeed file in the system (but repeating a TCF setting does no harm). You should set TCF before the LOGOUT is done. This can be done from a DLL as well as the EXE.

**Note:** We recommend using one transaction control file for a system. Using multiple TCF files with different access rights can result in partially committed transactions—some of the files within a transaction might be updated and others left unchanged.

**Note:** When you explicitly set the TCF, if the drive mapping is not the same on each workstation, then partially committed transactions can result.

## Supported Attributes and Procedures

|                                                                 |                  |                                                                 |                               |
|-----------------------------------------------------------------|------------------|-----------------------------------------------------------------|-------------------------------|
| <b>File Attributes</b>                                          | <b>Supported</b> |                                                                 |                               |
| CREATE .....                                                    | Y                | PACK( <i>file</i> ) .....                                       | Y <sup>6</sup>                |
| DRIVER( <i>filetype</i> [, <i>driver string</i> ]) .....        | Y                | POINTER( <i>file</i> ) .....                                    | Y <sup>8</sup>                |
| NAME .....                                                      | Y                | POINTER( <i>key</i> ) .....                                     | Y <sup>8</sup>                |
| ENCRYPT .....                                                   | Y                | POSITION( <i>file</i> ) .....                                   | Y <sup>9</sup>                |
| OWNER( <i>password</i> ) .....                                  | Y <sup>1</sup>   | POSITION( <i>key</i> ) .....                                    | Y <sup>9</sup>                |
| RECLAIM .....                                                   | N <sup>2</sup>   | RECORDS( <i>file</i> ) .....                                    | Y                             |
| PRE( <i>prefix</i> ) .....                                      | Y                | RECORDS( <i>key</i> ) .....                                     | Y                             |
| BINDABLE .....                                                  | Y                | REMOVE( <i>file</i> ) .....                                     | Y                             |
| THREAD .....                                                    | Y <sup>12</sup>  | RENAME( <i>file</i> , <i>new file</i> ) .....                   | Y                             |
| EXTERNAL( <i>member</i> ) .....                                 | Y                | SEND( <i>file</i> , <i>message</i> ) .....                      | Y                             |
| DLL( <i>flag</i> ) .....                                        | Y                | SHARE( <i>file</i> , <i>access mode</i> ) .....                 | Y                             |
| OEM .....                                                       | Y                | STATUS( <i>file</i> ) .....                                     | Y                             |
|                                                                 |                  | STREAM( <i>file</i> ) .....                                     | Y <sup>7</sup>                |
|                                                                 |                  | UNLOCK( <i>file</i> ) .....                                     | Y                             |
| <b>File Structures</b>                                          | <b>Supported</b> | <b>Record Access</b>                                            | <b>Supported</b>              |
| INDEX .....                                                     | Y                | ADD( <i>file</i> ) .....                                        | Y                             |
| KEY .....                                                       | Y                | ADD( <i>file</i> , <i>length</i> ) .....                        | N                             |
| MEMO .....                                                      | Y <sup>3</sup>   | APPEND( <i>file</i> ) .....                                     | Y                             |
| BLOB .....                                                      | Y                | APPEND( <i>file</i> , <i>length</i> ) .....                     | N                             |
| RECORD .....                                                    | Y                | DELETE( <i>file</i> ) .....                                     | Y <sup>2</sup>                |
| <b>Index, Key, Memo Attributes Supported</b>                    |                  | GET( <i>file</i> , <i>key</i> ) .....                           | Y                             |
| BINARY .....                                                    | Y <sup>13</sup>  | GET( <i>file</i> , <i>filepointer</i> ) .....                   | Y <sup>8</sup>                |
| DUP .....                                                       | Y                | GET( <i>file</i> , <i>filepointer</i> , <i>length</i> ) .....   | N                             |
| NOCASE .....                                                    | Y                | GET( <i>key</i> , <i>keypointer</i> ) .....                     | Y                             |
| OPT .....                                                       | Y                | HOLD( <i>file</i> ) .....                                       | Y                             |
| PRIMARY .....                                                   | Y                | NEXT( <i>file</i> ) .....                                       | Y                             |
| NAME .....                                                      | Y <sup>4</sup>   | NOMEMO( <i>file</i> ) .....                                     | Y                             |
| Ascending Components .....                                      | Y                | PREVIOUS( <i>file</i> ) .....                                   | Y                             |
| Descending Components .....                                     | Y                | PUT( <i>file</i> ) .....                                        | Y                             |
| Mixed Components .....                                          | Y                | PUT( <i>file</i> , <i>filepointer</i> ) .....                   | Y                             |
| <b>Field Attributes</b>                                         | <b>Supported</b> | PUT( <i>file</i> , <i>filepointer</i> , <i>length</i> ) .....   | N                             |
| DIM .....                                                       | Y                | RELEASE( <i>file</i> ) .....                                    | Y                             |
| OVER .....                                                      | Y                | REGET( <i>file</i> , <i>string</i> ) .....                      | Y                             |
| NAME .....                                                      | Y                | REGET( <i>key</i> , <i>string</i> ) .....                       | Y                             |
| <b>File Procedures</b>                                          | <b>Supported</b> | RESET( <i>file</i> , <i>string</i> ) .....                      | Y                             |
| BOF( <i>file</i> ) .....                                        | Y                | RESET( <i>key</i> , <i>string</i> ) .....                       | Y                             |
| BUFFER( <i>file</i> ) .....                                     | N                | SET( <i>file</i> ) .....                                        | Y                             |
| BUILD( <i>file</i> ) .....                                      | Y                | SET( <i>file</i> , <i>key</i> ) .....                           | Y                             |
| BUILD( <i>key</i> ) .....                                       | Y                | SET( <i>file</i> , <i>filepointer</i> ) .....                   | Y                             |
| BUILD( <i>index</i> ) .....                                     | Y                | SET( <i>key</i> ) .....                                         | Y                             |
| BUILD( <i>index</i> , <i>components</i> ) .....                 | Y <sup>15</sup>  | SET( <i>key</i> , <i>key</i> ) .....                            | Y                             |
| BUILD( <i>index</i> , <i>components</i> , <i>filter</i> ) ..... | Y <sup>15</sup>  | SET( <i>key</i> , <i>keypointer</i> ) .....                     | Y                             |
| BYTES( <i>file</i> ) .....                                      | Y                | SET( <i>key</i> , <i>key</i> , <i>filepointer</i> ) .....       | Y                             |
| CLOSE( <i>file</i> ) .....                                      | Y                | SKIP( <i>file</i> , <i>count</i> ) .....                        | Y                             |
| COPY( <i>file</i> , <i>new file</i> ) .....                     | Y                | WATCH( <i>file</i> ) .....                                      | Y                             |
| CREATE( <i>file</i> ) .....                                     | Y                | <b>Transaction Processing</b>                                   | <b>Supported<sup>10</sup></b> |
| DUPLICATE( <i>file</i> ) .....                                  | Y                | LOGOUT( <i>timeout</i> , <i>file</i> , ..., <i>file</i> ) ..... | Y <sup>11</sup>               |
| DUPLICATE( <i>key</i> ) .....                                   | Y                | COMMIT .....                                                    | Y                             |
| EMPTY( <i>file</i> ) .....                                      | Y                | ROLLBACK .....                                                  | Y                             |
| EOF( <i>file</i> ) .....                                        | Y                | <b>Null Data Processing</b>                                     | <b>Supported</b>              |
| FLUSH( <i>file</i> ) .....                                      | Y                | NULL( <i>field</i> ) .....                                      | N                             |
| LOCK( <i>file</i> ) .....                                       | Y <sup>5</sup>   | SETNULL( <i>field</i> ) .....                                   | N                             |
| NAME( <i>label</i> ) .....                                      | Y                | SETNONNULL( <i>field</i> ) .....                                | N                             |
| OPEN( <i>file</i> , <i>access mode</i> ) .....                  | Y                |                                                                 |                               |

## Notes

---

- 1 We recommend using a variable password that is lengthy and contains special characters because this more effectively hides the password value from anyone looking for it. For example, a password like “dd...#\$...\*&” is much more difficult to “find” than a password like “SALARY.”

**Tip:** To specify a variable instead of the actual password in the Owner Name field of the File Properties dialog, type an exclamation point (!) followed by the variable name. For example: !MyPassword.

- 2 The TopSpeed driver automatically reclaims space freed by deleted records and keys.
- 3 The TopSpeed file system uses the same compression algorithm for RECORDs and MEMOs. For data of 255 bytes or less, MEMOs have no disk space advantage over STRINGs. However, STRINGs are always allocated space (RAM) within the record buffer, whereas MEMOs are only allocated space when the file is OPENed. MEMOs do carry the advantage of BINARY versus NONBINARY, plus MEMOs may be omitted from all processing with the NOMEMO statement.
- 4 The TopSpeed driver does not support external names for keys, because all keys are stored internally.
- 5 LOCK() only affects other LOCK() calls. The only effect of a successful call to LOCK() is that other processes will get an error FLALLK when they call LOCK().

You must CLOSE the file when an ERRORCODE 90 occurs to release any locks that were in place when the error occurred.

- 6 PACK performs a BUILD and truncates the file to it's minimum size.
- 7 STREAM LOCKs the file against writes, but allows reads.
- 8 GET(*file,filepointer*) requires a pointer value returned from the POINTER() function. POINTER() returns a physical record address (not a record number). Therefore you cannot use

```
GET(file,1)
```

to retrieve the first record in a TopSpeed file because 1 is not a valid pointer in a TopSpeed file.

- 9 POSITION(*file*) returns a STRING(4). POSITION(*key*) returns a STRING the size of the key fields + 4 bytes.
- 10 TopSpeed file logging is very fast (about 100 times faster than the Clarion driver). With LOGOUT, the TopSpeed engine posts all transactions to memory. ROLLBACK simply frees the memory, while COMMIT writes out the database changes in a stream.

If a system crashes during a transaction (LOGOUT—COMMIT), the

recovery is automatically handled by the TopSpeed driver the next time the affected file is accessed.

- 11 LOGOUT LOCKs the file against writes, but allows reads. See also *PROP:Logout* in the *Language Reference*.
- 12 THREADED files do not consume additional file handles for each thread that accesses the file.
- 13 OEM conversion is not applied to BINARY MEMOs and BLOBs.
- 14 The TopSpeed driver accomplishes case insensitivity by converting strings to lowercase. This can cause unexpected behavior for characters that fall between the upper and lower case alphabet (that is, <sup>^</sup>(94) and <sub>\_</sub>(95) for both ANSI and ASCII sequences).
- 15 Dynamic indexes are built and kept in memory only until the file is closed.

## Other

### File Sharing

- ❖ SHARE and open access modes:

| <u>The following open access modes are supported</u> | <u>Share required</u> |
|------------------------------------------------------|-----------------------|
| 34 (12h) Read/Write, deny write (default for OPEN)   | Yes                   |
| 66 (42h) Read/Write, deny none (default for SHARE)   | Yes                   |
| 64 (40h) Read Only, deny none                        | Yes                   |
| 18 (12h) Read/Write, deny all                        | No                    |
| 16 (10h) Read Only, deny all                         | No                    |
| 32 (20h) Read Only, deny write                       | No                    |

For the modes indicated, SHARE.EXE (which implements DOS record locking) must be loaded in AUTOEXEC.BAT or CONFIG.SYS. The following example loads SHARE in AUTOEXEC.BAT, providing 500 maximum file locks, and the default 2048 bytes for the storage area.

```
C:\DOS\SHARE.EXE /L:500
```

If SHARE.EXE is required but is not loaded when the driver tries to obtain a lock, the program generates a run-time message of "Failed to re-open in exclusive mode."

**Tip:** You do not need SHARE.EXE (or VSHARE) in any environment (for example, Novell, Win95 or Win NT) that supplies file locking as part of the operating system.

If there is no form of SHARE present (SHARE or VSHARE), then, for the first file access, the driver opens the file in exclusive mode. Thereafter, subsequent attempts to open the file will fail.

### ERRORCODE 90

The TopSpeed driver posts an ERRORCODE of 90 for unexpected runtime errors. At the same time, the driver posts a FILEERRORCODE (the former TPSBT error code) that helps us diagnose the problem. This error handling gives you more control over runtime errors and provides us with more information. That is, your program can trap for ERRORCODE=90 and react accordingly.

You must CLOSE the file when an ERRORCODE 90 occurs to release any locks that were in place when the error occurred.

Should you receive an ERRORCODE of 90 from the TopSpeed driver, we want to know about it. Please send us a copy of the file and the corresponding FILEERRORCODE value.

## **Large Keys (or small RAM)**

APPEND() is recommended over ADD() if the total size of the keys exceeds the amount of RAM available, if there is more than one key, or when adding a large number of records. The size of a key (for this purpose) is the number of entries times (the sum of key fields + 10 bytes). If the records being added are already in an approximate key order, then you can discount that key for the purposes of the above calculation.

As an example, if a file has two 40 byte keys and 2 Megabytes of RAM are available, then ADD() becomes (relatively) slow when the database size exceeds about  $2,000,000 / (40 + 10 + 40 + 10) = 20,000$  records.

## **Incremental Key/Index Build**

The TopSpeed driver implements incremental building; this means that building a key only reads records starting from the first record appended since the key was last built. The driver merges the new keys with the existing key. Thus building a large key where only a few recently added records have been modified should be *fast*. See the FULLBUILD driver string above.

Building an index is similar, but must start at the minimum physical record whose position in the index has changed since the index was last built.

Dynamic indexes are not retained, so cannot be built incrementally.

## **Batch Processing Performance**

When *writing* a large number of records, use STREAM() or open the file in a deny write mode, that is, OPEN(file) rather than SHARE(file). After the records have been written, call FLUSH() to allow other users access.

It is very important to use STREAM() when ADDing/APPENDing/PUTting a large number of records. STREAM() will typically make processing about 20 times faster. For example, adding 1000 records might take nearly 2 minutes without STREAM(), but *only 5 seconds* with STREAM.

It is not necessary to use STREAM() or FLUSH() on a logged out file (performance on logged out files is always good).

STREAM has the effect of LOCKing the file.

## **POINTERS and Deleted Records**

POINTER(key) returns the relative position of the record within the file. Consequently when that record is DELETED, the pointer becomes invalid. Any subsequent access using the pointer fails. If you require fuzzy matching whereby the nearest record is returned, use the POSITION() function.

## Data Compression—STRINGS v MEMOs

The TopSpeed driver compresses the entire record buffer area (not individual fields within the record), therefore, compression gains can be realized by placing similar fields adjacent to each other in the FILE declaration.

The TopSpeed file system uses the same compression algorithm for STRINGS and MEMOs; however, the compression occurs at a “higher level” for MEMOs than for STRINGS. As a result, MEMOs do have a disk space advantage over large STRINGS (over 500 bytes) and smaller STRINGS can have a slight performance advantage over MEMOs. The larger the STRING, the greater the advantage.

MEMOs do carry the advantage of BINARY versus NONBINARY, plus MEMOs may be omitted from all processing with the NOMEMO statement.

STRINGS are always allocated space (RAM) within the record buffer, whereas MEMOs are only allocated space when the file is OPENed. Also MEMOs cannot be key components.

## Estimating File Size

The TopSpeed file driver compresses data and key information, so the ultimate file size depends on the “compressibility” of the data and keys. In the worst case (data and keys cannot be compressed because there is no repeating information) the file size may be estimated as:

$$(\text{RecordSize} + \text{All Key components}) * \text{Records} + \text{Fixed Overhead}$$

In a more realistic case (data and keys are compressible), the file size may be estimated as:

$$((\text{size of all string fields})/(\text{compressibility factor}) + \text{size of all binary fields} + \text{size of all binary key components} + (4 * \text{number of string key components})) * \text{Records} + \text{Fixed Overhead}$$

Note that Fixed Overhead varies depending on your file definition. Fixed overhead includes about 800 bytes for the driver, plus the header information describing the fields and keys for the file. The more fields and keys, and the longer the names, the higher the fixed overhead. A rough rule of thumb for calculating fixed overhead is 800 bytes + 40 bytes for each field and key. For Example:

| File Description    | Estimated Fixed Overhead |
|---------------------|--------------------------|
| 1 field, no keys    | 1KB                      |
| 20 fields, 10 keys  | 2KB                      |
| 200 fields, 10 keys | 9KB                      |

## **Concurrent User Limit**

The TopSpeed driver limits concurrent users to 1024 per file; additional users would have to wait momentarily until a slot opens up. Practically speaking the driver is very unlikely to reach this limit since very few networks and servers will support this many concurrent users. Generally, we recommend a client/server file system for more than 30 concurrent users.

## **Transaction Processing—the TCF File**

---

### **Speedy Logging and Automatic Recovery**

TopSpeed transaction logging is very fast (about 100 times faster than the Clarion driver). With LOGOUT, the TopSpeed engine posts transactions to a safe area on disk. ROLLBACK simply flags the transaction as incomplete, while COMMIT makes the disk areas a valid part of the TopSpeed files and marks the updates as finalized on disk (not to be rolled back).

If a system crashes during a transaction, the TopSpeed driver automatically handles the recovery the next time the affected file is accessed.

If the COMMIT is not done then the updates are rolled-back. Note that for RI purposes you can read the uncommitted updates before the COMMIT takes place.

### **Transaction Control File**

The TopSpeed file system uses a transaction control file (TCF) to ensure that changes to more than one DOS file, which are grouped into a transaction, either all happen or none happen. The TCF provides a single (boolean) storage location that indicates whether a multi-file transaction committed or not. The TCF contains very little information; it just serves to coordinate multi-file commits. The actual rollback/commit data is stored in the data (.TPS) files.

### **Transaction Control File Location**

When any workstation finds a TopSpeed file that is in a partially committed state, and that was involved in a multi-file transaction, it needs to access the transaction control file (TCF) to decide what to do. To be effective, the TCF must be accessible when any files controlled by it are accessed by any workstation. Therefore, you generally should use a single TCF per system, and you should not delete or move the TCF.

By default the TCF pathname is “*datafilepath*\TOPSPEED.TCF,” where *datafilepath* is the path of the TopSpeed data file. There is a single TCF (Topspeed.TCF) in each directory containing TopSpeed files that are logged out. This default assumes a “typical” system where all data files are in a

single directory. If your datafiles are not in a single directory, you can set the TCF location with the TCF driver string so that your system uses only one TCF. The TCF setting is a global one—it applies to all TopSpeed files accessed by the program. See *TCF* for more information.

**Tip:** Clarion 4 and earlier versions used `root:\TOPSPEED.TCF` as the default TCF Location. However, the TCF location is calculated at the start of the commit phase and is stored in the TopSpeed file, so the change to `datafilepath\TOPSPEED.TCF` is backward compatible.

It is possible to use multiple TCF files within a system; however, we strongly discourage this. The consequence of there being several TCF files with various levels of accessibility (or of a deleted or overwritten TCF file) is that some of the files within a transaction might be updated and others left unchanged.

### How TopSpeed Transaction Logging Works

LOGOUT gives each transaction a unique id which it stores in the transaction control file (TCF). LOGOUT also stores the TCF file name and transaction id in each data (.TPS) file which is updated, so that after a crash, the next time the file is opened the TopSpeed driver can find the TCF and do any necessary recovery. COMMIT removes the unique transaction id from the TCF.

If the COMMIT is not done then the updates are rolled-back. Note that for RI purposes you can read the uncommitted updates before the COMMIT takes place.

## Storing Multiple Tables in a single .TPS File

By using the characters '\!' in the NAME() attribute of a TopSpeed file declaration, you can specify that a single .TPS file will hold more than one table. For example, to declare a single .TPS file 's&p.tps' that contains 3 tables called *supp*, *part* and *ship*:

```
Supp FILE,DRIVER('TopSpeed'),PRE(Supp),CREATE,NAME('S&P\!Supp')
...
Part FILE,DRIVER('TopSpeed'),PRE(Part),CREATE,NAME('S&P\!Part')
...
Ship FILE,DRIVER('TopSpeed'),PRE(Ship),CREATE,NAME('S&P\!Ship')
...
```

The tables share a single DOS file handle, opened when the first table is opened, and closed when the last table is closed. The first open mode determines the open mode for *all* the other tables in the file. If the first open mode is read-only, then all the tables are read-only and no updates are allowed.

Similarly, if one of the tables in the file is logged out, then all the tables are effectively logged out. If one table in the file is flushed, then all the tables are flushed.

This feature is especially useful when there are a large number of small tables, or when the application must normally access several related tables at once.

You can retrieve the names of tables within the .TPS files with the SEND() command. To retrieve the first name, issue:

```
SEND(file, 'PNM=')
```

This returns the name of the first table. To retrieve the second name, issue:

```
SEND(file, 'PNM=FirstTableName')
```

This returns the name of the second table, and so on.

You can also rename the tables; for example, given the above declarations the following command renames the table called Supp to Old\_Supp:

```
RENAME(Supp, 'S&P\!01d_Supp')
```

If you use the OWNER attribute on multiple tables in a single .TPS file, all the tables must have the same OWNER attribute.

If you don't specify a table name, the table is called 'unnamed', so that the following are all equivalent:

```
foo FILE, DRIVER('TopSpeed')
foo FILE, DRIVER('TopSpeed'), NAME('foo')
foo FILE, DRIVER('TopSpeed'), NAME('foo\!unnamed')
```

## Collating Sequences

---

Changing the collation sequence on a Clarion 2.003 or earlier TopSpeed file (by changing .ENV file or OEM flag) corrupts the file.

With Clarion 4, this is no longer true because the collating sequence for the file is now stored within the file. This change is fully backward compatible. Old files continue to work as before and new files are accessible by older programs.

To add the collating sequence information to an existing file, simply do a full build on the file:

```
SEND(file, 'FULLBUILD=on')
BUILD(file)
```

The collating sequence for a TopSpeed file is established when the table is created or a full build is performed. Therefore the OEM flag is only significant at the creation of the file or on a full build.

Any application that uses an incorrect sequence (due to an incompatible .ENV file) to access a file may get unpredictable results, but will not corrupt the data.

## Accessing TopSpeed files with Access Jet and ODBC

---

Occasionally, the Access Jet engine returns “#deleted” for each requested field. This is a known bug in Microsoft Access. The default query used by Access does an internal comparison of the record set to determine if a record has been deleted or modified from the database. The mechanism is known to work poorly for certain data types, notably DECIMAL.

Microsoft recommends using an SQL pass-through query as a work around to this problem. To create a SQL pass-through query:

1. Choose SQL Specific, Pass Through from the Query menu in query design mode. For Access 7, Select Query, and press the New button.
2. Accept the default of Design View.
3. Close the Show Table Window.
4. Select SQL Specific from the Query menu and select the Pass-Through option.
5. Enter the SQL statement.

The Query can be saved for future use. This method will correct virtually all display problems, but the resulting grid is not updatable. Updates must be performed using an Update Query when SQL Passthrough is used.

## ***TopSpeed Database Recovery Utility***

The TopSpeed file system is designed to automatically repair most errors. However, if a TopSpeed file is physically damaged during a system malfunction, the TopSpeed Database Recovery Utility can recover the undamaged portions of your data.

**Note:** The TopSpeed Database Recovery Utility is an emergency repair tool and should not be used on a regular basis. Use it only when a file has been damaged.

The TopSpeed Database Recovery Utility reads the damaged file and writes the recovered records to a new file. It uses the information stored in the file's header and scans the file recovering undamaged portions.

Optionally, you can provide an example file containing the header information in the event the original header information is damaged. An example file is any file with a FILE declaration identical to the damaged file. You can create an example file by issuing a CREATE(file) command, then saving the resulting empty file to a new name.

The TopSpeed Database Recovery Utility is a distributable utility designed to help your end users recover damaged files.

**Tip:** The Clarion license agreement applies to TPSFIX.EXE. You may distribute to your users, but they may not redistribute it.

The recovery utility is designed to work either interactively or noninteractively with command line parameters. Interactively, you provide the parameters through two wizard dialogs. You can run TPSFIX noninteractively by supplying the command line parameters with the Clarion RUN() statement, Windows API calls, Windows 95 shortcuts, or Program Manager Icons.

### **ERRORCODE 90 and Corrupted Files**

---

The TopSpeed driver posts an ERRORCODE of 90 for unexpected runtime errors. When an ERRORCODE of 90 occurs, the driver also posts a FILEERRORCODE (the former TPSBT error code) that helps us diagnose the problem.

An ERRORCODE of 90 usually indicates your TopSpeed file is corrupted. In most cases the corruption is a result of hardware failure. For example, one customer with a 50 machine network traced a near daily file corruption to bad network cards on 2 of the 50 machines. After replacing the bad cards, the corruptions disappeared.

However, should you receive an **ERRORCODE** of 90 from the TopSpeed driver, we want to know about it. Before you repair the file, please make a copy of the damaged file and send it to us along with the corresponding **FILEERRORCODE** value. We analyze all the corrupted files we receive for recognizable patterns that can help us improve the driver.

## Using the Recovery Utility Interactively

---

1. Start the utility by **DOUBLE-CLICKING** on the TopSpeed Database Recovery Utility Icon In the Clarion Program Group.

The **TopSpeed Database Recovery Utility** dialog appears. The utility consists of two wizard dialogs.

2. In the **Source** (file to recover) section, specify the file name or press the **Browse** button to select it from a standard file open dialog.

3. If the file has a password, type it in the **Password** entry box.

If the database file contains multiple tables (data files), each table *must* have the same password.

4. Optionally, in the **Destination** (result file) section, specify the file name for the target file or press the **Browse** button to select it from a standard file open dialog.

By default the **.TPR** extension is added to the source file name. This parameter is optional. If omitted, the original (source file) is overwritten and a backup file is created. The source file is renamed to *filename.TPx*, where x is automatically incremented from 1 to 9 each time a new file is created. If all nine numbers are used, any subsequent files created are given the extension **.TP\$** and are overwritten.

5. If the result file is to have a different password, type it in the **Password** entry box. If omitted, the password is removed.

6. Press the **Next** button.

The second wizard dialog for the TopSpeed Database Recovery Utility appears.

7. Optionally, specify the **Example File** file name or press the **Browse** button to select it from a standard file open dialog.

The utility uses the Example File to determine table layouts and key definitions in the event those areas of the source file are damaged. The default extension is **.TPE**, but if you choose, you may use any valid DOS extension

**Tip:** We recommend shipping an example file when you deploy your application. This improves data recovery from a damaged file.

8. If the example file has a password, type it in the **Password** entry box.

9. If you want the utility to rebuild Keys, check the **Build Keys** box.  
If omitted, the keys are rebuilt by the original application when it attempts to open it.
10. If you want to use the Header Information in the source file, check the **Use Header** box.  
Using Header Information optimizes the utility's performance, but should not be used if the file header is corrupt. If omitted, the utility searches the entire data file and restores all undamaged pages.
11. If the application uses a Locale (.ENV) File for an alternate collating sequence, specify the .ENV file or press the **Browse** button to select it from a standard file open dialog.
12. If the file is using the OEM attribute to control the collating sequence, Check the **Use OEM** box.  
This enables the OEMTOANSI and ANSITOOEM conversion.
13. Press the **Start** button to begin the recovery process.  
If the utility does not find any errors, a message appears informing you that "No Errors Detected in <filename.ext>" and asks if you want to continue with recovery.

## Command Line Parameters

---

The TPSFIX utility can accept command line parameters which lets you execute it from an application, from a Program Manager Icon, or from a Windows 95 Shortcut.

Here is the syntax for running TPSFIX noninteractively with command line parameters.

```
TPSFIX sourcepath[?password] [destpath[?password]] [/E:examplepath[?password]]
 [/L:localepath] [/H] [/K] [/P] [/O]
```

|                        |                                                                                                                                                                      |
|------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>TPSFIX</b>          | The executable (TPSFIX.EXE).                                                                                                                                         |
| <i>sourcepath</i>      | The file name and path of the source (damaged) database file.                                                                                                        |
| <i>?password</i>       | The file's password.                                                                                                                                                 |
| <i>destpath</i>        | The file name and path of the recovered database file. If omitted, the <i>destpath</i> is the same as the <i>sourcepath</i> and an example file is required.         |
| /E: <i>examplepath</i> | The file name and path of the example database file. This parameter is required for any fix-in-place operation (that is, when <i>sourcepath</i> = <i>destpath</i> ). |
| /L: <i>localepath</i>  | The file name and path of the Locale (.ENV) file used to                                                                                                             |

|                 |                                                                                                                                                             |
|-----------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                 | specify an alternate collating sequence.                                                                                                                    |
| <code>/H</code> | If specified, the utility uses the header information in the source file.                                                                                   |
| <code>/K</code> | If specified, the utility rebuilds all keys for the database.                                                                                               |
| <code>/P</code> | If specified, the user is prompted for each parameter even if they are supplied on the command line.                                                        |
| <code>/O</code> | If specified, the file uses OEMTOANSI and ANSITOOEM to determine the collating sequence. See <i>Internationalization</i> in the <i>Language Reference</i> . |

## Using the Recovery Utility Non-Interactively

---

There are some issues to consider before running the TPSFIX utility. Because of the following, we do not recommend running TPSFIX from your application program. Rather, it is better to instruct your end users to close down the application program completely before running the TPSFIX utility.

- The database file should NOT be open when running TPSFIX. Ensure the file is closed before starting TPSFIX.
- To prevent access during the recovery process is completed, TPSFIX LOCKs the file automatically.
- It is more efficient and safer to have your application rebuild the KEYS (omit the /K parameter). It is also a good way to check the status of a recovery.

### Automatic Fix-in-Place Recovery

By omitting the *destpath* parameter and supplying an example file, you can directly overwrite the damaged file. This is a fix-in-place recovery. The TPSFIX utility does create an intermediate file, but you don't have to worry about it. For Example:

```
TPSFIX.EXE Datafile.TPS /E:Example.TPE /H
```

or with Embedded Source Code:

```
RUN('TPSFIX.EXE Datafile.TPS /E:Example.TPE /H')
```

This recovers the "datafile.TPS" file using the "Example.TPE" file as an example for the table and key layouts, does not rebuild the keys, and uses the header information in the original file. TPSFIX automatically saves the original file to a backup with a file extension of TP1 through TP9. Each time the utility is executed, the numeric portion of the extension is incremented.

## **Separate Source and Target Recovery**

This method requires two lines of embedded source code but gives you control over the renaming process. You insert the source code in the Accepted Embed point for the Menu Item or button. For example:

```
COPY(datafilelabel, 'Datafile.OLD') ! copies the original file
 ! to Datafile.OLD
RUN(TPSFIX Datafile.OLD Datafile.tps /H)! Runs the utility using the
 ! renamed file as
 ! the source and the original
 ! name as the target
```

This copies the datafilelabel file to DATAFILE.OLD, recovers the file and writes it to DATAFILE.TPS using the header information in the original file.

## TopSpeed Database Copy Utility

The TopSpeed Database Copy Utility lets you copy files into and out of TopSpeed Database files which contain multiple tables (data files) in a single DOS file.

**Tip:** The Clarion license agreement applies to TPSCOPY.EXE; it is not freely distributable. You may distribute to your users, but they may not distribute it.

### Example Uses

---

Using this utility you can:

- ◆ Extract files from a multi-table TopSpeed Database. This lets you extract a file, run a conversion program, then reinsert it.
- ◆ Insert files into a multi-table TopSpeed Database. This lets you create each file separately, then insert them all into a single .TPS file.
- ◆ Replace files in a multi-table TopSpeed database.
- ◆ Remove files from a multi-table TopSpeed database.
- ◆ Add a new table to a multi-table TopSpeed database.
- ◆ Add a new key to an existing file in a multi-table TopSpeed database.
- ◆ Add a new field or modify an existing field in a table within a multi-table TopSpeed database.
- ◆ Remove or modify a field or key in a table in a multi-table TopSpeed database.
- ◆ Combine single-table TopSpeed database files into a single multi-table TopSpeed Database file.
- ◆ Copy a data file in a multi-table TopSpeed database file to another database.

### File Sharing Considerations

---

**Note:** Before using this or any utility on “live” data files, you should back up your files.

The database file should NOT be open when running the utility. Ensure that the file is closed before starting the utility. If exclusive access is denied, the utility displays an “Access Denied” warning and disables the **Copy** button.

To prevent access until the copy process is completed, the utility locks the file automatically.

## Copy Utility Interface

---

### Source (copy from)

- Database Name** The DOS filename for the source TopSpeed database file from which tables are copied. Type in a filename, or select an existing file by pressing the **Browse** button.
- Password** The source file's password (OWNER attribute). If the file has a password, it must be supplied. If the source database has no password, this entry box is disabled.
- File to Copy** The table within the TopSpeed Database from which to copy. Select from a drop down list. If the database contains only one table, it is usually named UNNAMED.
- Remove File from Source Database** Check this box if you want to delete the table from the source database after it is copied. If not checked, the table remains in the source database. When using this option, when the last table is removed from the source database, you have the option to delete the physical file from disk.
- Rebuild Keys** Do a full BUILD of the source database keys.

### Destination (copy to)

- Database Name** The DOS filename for the target TopSpeed database file to which a table is copied. Type in a filename (including an extension) to create a new file, or select an existing file by pressing the **Browse** button.
- Password** The password (OWNER attribute) for the destination database. If the destination database has a password, it must be supplied. If the destination database has no password, and one is supplied, it is added. If a new database is being created, enter a new password here.
- New File** The name of the Table within the database to which the source file is copied. This defaults to the table name from the source database. If you want to create a single-table database, the New File should be named UNNAMED.

|             |                                                                |
|-------------|----------------------------------------------------------------|
| <b>Copy</b> | Press this button to copy the source table to its destination. |
| <b>Exit</b> | Press this button to close the utility.                        |

## Copy Utility Command Line Parameters

```
TPSCOPY sourcefile[?password]!tablename destinationfile[?password]!tablename [/R] [/D]
```

|                        |                                                                                                                                                      |
|------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>TPSCOPY</b>         | Start the executable (TPSCOPY.EXE)                                                                                                                   |
| <i>sourcefile</i>      | The DOS filename for the source TopSpeed database file from which tables are copied                                                                  |
| <i>?password</i>       | The file's password (OWNER attribute). If the file has a password, it must be supplied. If the database has no password, this can be omitted.        |
| <i>!tablename</i>      | The name of the specific table within the TopSpeed Database. If the database contains only one table, it is named UNNAMED.                           |
| <i>destinationfile</i> | The DOS filename for the target TopSpeed database file to which a table is copied. Type in a filename (including an extension) to create a new file. |
| <i>/R</i>              | Remove the table from the source database after it is copied.                                                                                        |
| <i>/D</i>              | Delete a source database which contains no remaining tables (remove the physical file from disk).                                                    |
| <i>/K</i>              | Do a full BUILD of the source database keys prior to copying.                                                                                        |

### **Examples:**

```
TPSCOPY orders.tps?acme!CUSTOMER customer.tps?acme96!UNNAMED
```

Copies the customer table from ORDERS.TPS (using the password acme) to a single-table file CUSTOMER.TPS which has the password acme96.

```
TPSCOPY orders.tps?fred!CUSTOMER customer.tps?barney!UNNAMED
```

Copies the customer table from ORDERS.TPS (using the password fred) to a single-table file CUSTOMER.TPS which has the password barney.

```
TPSCOPY customer.tps!UNNAMED orders.tps?acme96!CUSTOMER
```

Copies the UNNAMED (the only table in the file) table from CUSTOMER.TPS (which has no password) to the CUSTOMER table in ORDERS.TPS which has the password acme96.

# PART V

---

## SQL ACCELERATOR DRIVERS



# 34 - SQL ACCELERATORS

## Overview

TopSpeed's SQL Accelerator Drivers include

- AS400
- MSSQL
- ODBC
- Oracle
- Pervasive (Scalable) SQL
- SQL Anywhere

These SQL Accelerator Drivers share a common code base and many common features such as TopSpeed's unique, high speed buffering technology (see *BUFFER* in the *Language Reference*), common driver strings, and SQL logging capability. However, their primary purpose is to translate Clarion file commands into appropriate, efficient SQL statements specific to their respective SQL servers, and to handle any result sets returned by those servers.

**Tip:** For best performance, we strongly recommend using the ABC Templates to generate your SQL based applications.

## TopSpeed's SQL Accelerators

---

TopSpeed's SQL Accelerators support the common driver features documented in *Part I*. See *Part I—Database Driver Overview* for more information.

The SQL Accelerators convert standard Clarion file I/O statements and function calls into optimized SQL statements, which they send to their backend SQL servers for processing. This means you can use the same Clarion code to access both SQL tables and other file systems such as TopSpeed files. It also means you can use Clarion template generated code with your SQL databases.

In addition to the automatically generated SQL statements, the SQL Accelerators forward any additional SQL statements you specify to the backend SQL servers. The SQL Accelerator interprets the result set returned from the SQL server and makes it available to your application program with the Clarion NEXT or PREVIOUS statement.

All the common behavior of all the SQL Accelerators is documented in this chapter. Driver-specific behavior is documented the chapter for that specific SQL driver.

## Unique Keys

---

The SQL Accelerator drivers should generally be used only on tables with unique keys. The drivers will function on files without unique keys, but only with substantially limited capabilities. Without a unique key, the RESET and REGET commands return errors, and the driver cannot update the SQL database.

Most Clarion templates also require that you define a primary key for each table in order to generate code.

## ABC Templates and SQL

---

The ABC Templates are more “SQL Friendly” than the Clarion Templates. That is, the ABC Templates are more efficient (they refresh from disk only when absolutely necessary or when explicitly requested to do so); they can be optimized for use with SQL databases (see *Classes Tab Options—Global—BrowseClass Configuration*); they take advantage of TopSpeed’s Accelerator technology (see *BUFFER in the Language Reference* for more information); and they allow file-loaded browses which eliminate costly backward paging (see *Control Templates—BrowseBox*).

The ABC Templates are more strict in their implementation of some Data Dictionary settings. In particular, the Clarion Templates ignore the “case insensitive key” setting for SQL databases, but the ABC Templates do not.

**Note:** Use caution when clearing the Case Sensitive box in your SQL based Data Dictionary. Unlike the Clarion Templates, the ABC Templates enforce caseless keys which can drastically slow down your application.

The ABC Templates apply the SQL UCASE(keyfield) command if you request “caseless” keys. Most SQL backends ignore keys and do a full table scan when a key element is the subject of a scalar function such as UCASE. Therefore, clearing the **Case Sensitive** box in your Data Dictionary results in a UCASE(keyfield) in your generated SQL Statement which forces a full table scan (ignoring any indexes) and drastically slows down your application.

## Using SQL Tables in your Clarion Application

### Register the SQL Accelerator

---

Before your application can use a particular database driver, the driver must be registered with the Clarion development environment. The in-the-box drivers are already registered when you install Clarion. You must register any add-on drivers. See *Clarion's Development Environment—Database Driver Registry* in the *User's Guide* for information on registering database drivers.

### Import the Table Definitions

---

Typically, you add SQL support to your application by importing the SQL table, view, and synonym definitions into your Clarion Data Dictionary. See *The Dictionary Editor—Importing File Definitions* in the *User's Guide* for general information on importing table, file, and view definitions. This section describes SQL Driver imports generally. Driver-specific import information is described in the chapter or manual for each driver.

**Tip:** To import table definitions, you must have read privileges for the SQL database's system tables—except for security or access tables (user ids, passwords, privileges, etc.)

Although you can manually add table definitions to the dictionary (or even hand code your FILE declarations) for your SQL tables, we strongly recommend importing the table definitions. Importing the table definitions reduces the chance of introducing errors into the dictionary and guarantees the correct specification of data types, key structures, etc.

The importing approach assumes your SQL tables are already defined within the SQL database. In the case where you are designing a new SQL database, you may, of course, lay out the table definitions for the first time in the Clarion Data Dictionary. However, we recommend this approach only for prototyping and for databases with minimum complexity and maintenance requirements. In most cases, to correctly implement an SQL database requires defining more items than are stored in the Clarion Data Dictionary—for example, stored procedures, triggers, access rights, and storage allocation.

Once your table definitions are in the Clarion Data Dictionary, you develop your SQL based applications just as you would any other application.

**Note:** Driver-specific import information is described in the chapter or manual for each driver.

## SQL Import Wizard—Login Dialog

---

When you select an SQL Accelerator Driver from the driver drop-down list, the Import Wizard opens the **Login/Connection** dialog. The **Login/Connection** dialog collects the connection information for the SQL database.

**Note:** Before you can connect to the SQL database and import table definitions, the database must be started and must be accessible from your computer.

Fill in the fields in the **Login/Connection** dialog.

**Next >**

Press this button to open the Import Wizard's **Import List** dialog.

## SQL Import Wizard—Import List Dialog

---

When you press the **Next >** button, the Import Wizard opens the **Import List** dialog. The **Import List** dialog lists the importable items.

Highlight the table, view, or synonym whose definition to import, then press the **Finish** button to import. The Import Wizard adds the definition to your Clarion Data Dictionary, then opens the **File Properties** dialog to let you modify the default definition.

Import additional tables, views, and synonyms by repeating these steps. After all the items are imported, return to the Dictionary Editor where you can define relationships and delete any columns not used in your Clarion application. See *Advanced Techniques—Define Only the Fields You Use*.

## Connection Information and Driver Configuration—File Properties

---

Typically, you add SQL support to your application by importing the SQL or ODBC table, view, and synonym definitions into your Clarion Data Dictionary. The Import Wizard automatically fills in the **File Properties** dialog with default values based on the imported item. However, there are several fields in the **File Properties** dialog you can use to further configure the way the SQL Accelerator Driver accesses the data. These **File Properties** fields are described below.

### Driver Options

Typically, the Import Wizard places nothing in the **Driver Options** field. However, you can add driver strings to this field to control how the driver accesses your SQL data. For example, you can generate a log of driver

activity or specify how the driver handles dates with a value of zero (0). See *SQL Driver Strings* for more information.

### **Owner Name**

Typically, the Import Wizard places the SQL database connection information (Host, Username, Password, etc.) in the **Owner Name** field.

For security and portability reasons, you may want to specify this connection information with variables rather than hard coded strings in your dictionary. To use a variable specification, type the variable name, preceded by an exclamation point in the **Owner Name** field; for example !LoginString. Then use whatever method you choose to prime the variable before accessing the SQL table.

Some SQL Accelerators allow additional information in the **Owner Name** field. This information is described in the chapter for each driver.

## SQL Driver Behavior

### Automatic Login Dialog

---

The SQL Accelerator drivers automatically look for UserName and Password values whenever they access an SQL table. If a UserName and Password have already been supplied, the driver uses those values. If no values have been supplied, the driver optionally prompts for the UserName and Password with the automatic login dialog. See *PROP:LogonScreen* for the specific driver for more information.

We recommend opening a table at the start of your program so the time devoted to logging in occurs at program start up. Clarion's Application Wizard automatically generates code to do this for SQL based programs. However, if you do not use the Application Wizard, you can accomplish the same effect simply by adding an SQL table to the File Schematic for your main procedure. This automatically generates code to open the table.

### SET/NEXT and SET/PREVIOUS Processing (SELECT/ORDER BY)

---

A SET statement followed by a NEXT in a LOOP structure is the most common Clarion method to process records sequentially. When the SQL Accelerator drivers encounter a SET/NEXT combination, they generate an SQL SELECT statement with an ORDER BY clause based on the KEY component fields. The KEY component fields are determined by the KEY names in the SET statement. For example, the SQL driver translates this Clarion code

```

Ord FILE,PRE(Ord),DRIVER('SQLDriver'),NAME('ord')
NameDate KEY(+Ord:Name,+Ord:Date),NAME('DateKey')
Record RECORD
Name STRING(12),NAME('NameId')
Date DATE,NAME('OrderDate')
Type STRING(1),NAME('OrderType')
Details STRING(20),NAME('OrderDetails')
. . .
CODE
Ord:Name = 'SMITH'
SET(Ord:NameDate,Ord:NameDate)
LOOP
 NEXT(Ord)
 !... some processing
END

```

into a SELECT statement similar to:

```

SELECT NameId,OrderDate,OrderType,OrderDetails FROM Ord
WHERE (NameID >= 'SMITH')
ORDER BY NameID, OrderDate

```

**Tip:** The `SET(file)` statement (to process in file order, not keyed order) only supports the `NEXT` statement. Any attempt to execute a `PREVIOUS` statement when processing in file order causes `ERRORCODE 80 (Function Not Supported)`.

## Null Fields

---

### ISNULL, SETNULL, SETNONULL

When you read a row with NULL values from an SQL table, the Clarion record buffer contains an empty string for string fields, or a 0 for numeric fields, and `ISNULL(field)` returns `TRUE` for the field. If the field's contents are later changed to a non-empty or non-zero value then `ISNULL(field)` returns `FALSE`.

If you want to change a NULL field to non-null, but still blank or zero, then you must call `SETNONNULL(field)` to reset the null flag.

If you wish to clear a field to NULL that was previously non-null then call `SETNULL(field)` or `SETNULL(record)`. `SETNULL()` clears the contents of the field or record and resets the null flag.

When adding a new record to a file, by default all blank fields are added as blank or zero fields, not as NULL. If you want to force a field to be added with a NULL value, then you must call `SETNULL(field)` or `SETNULL(record)` to null all the fields.

**Tip:** The ABC Templates provide global `PrimeRecord` and `ValidateField` embed points where you can handle NULL settings for the entire application.

### Filtering on Nulls

You can use the SQL Accelerators to filter on null values using the following special syntax.

**Tip:** If you use `PROP:SQLFilter`, you can use the native SQL `ISNULL` or `IS NOT NULL` syntax.

The SQL generator converts a Clarion filter value of `NULL(field)` to `IS NULL` syntax using the following rules:

| Clarion Filter                      | Generated SQL Filter           |
|-------------------------------------|--------------------------------|
| <code>NULL(field) = 0</code>        | <code>field IS NOT NULL</code> |
| <code>NULL(field) = non-zero</code> | <code>field IS NULL</code>     |

NULL(*field*) <> 0            *field* IS NULL  
NULL(*field*) <> non-zero    *field* IS NOT NULL

For example

```
MyView{PROP:Filter}='NULL(CUS:ID)=0'
```

is equivalent to

```
MyView{PROP:SQLFilter}='Sales,CUS.ID IS NOT NULL'
```

## Performance Considerations

Generally, Clarion's development environment (Data Dictionary Import Wizard, Database Drivers, and templates) produce optimized, high performance, SQL applications.

**Tip:** For best performance, we strongly recommend using the ABC Templates to generate your SQL based applications.

This section describes some of the issues involved in producing these optimized applications. You should be aware of these issues so you can maintain a high level of performance as you take more control of the development process.

### Define Only the Fields You Use

---

With the SQL Accelerator drivers you only need to define the fields that you actually use in the Clarion Data Dictionary. This reduces both the overhead within your Clarion application and network traffic.

For example, if your SQL table contains 200 columns but only three are needed for a particular program, retrieving only those three fields dramatically reduces the amount of data sent over the network. If each column contains 20 bytes, then three columns would require only 60 bytes to be transferred whereas all 200 columns requires a 4,000 byte transfer.

After you have imported the table definition into your Clarion Data Dictionary, use the Dictionary Editor's **Field / Key Definition** dialog to delete the fields/columns you don't use.

### Matching Clarion Keys to SQL Constraints and Indexes

---

Generally, the Clarion KEY definition need not exactly match an index in the SQL database. The Clarion KEY simply serves to supply the appropriate ORDER BY clause for driver generated SELECT statements.

However, if the Clarion KEY does not match an SQL key or index, then the SQL server must build a temporary logical view every time you access the table using the unmatched KEY. This can be very slow for large files.

The best way to guarantee the Clarion KEYs have a matching SQL constraint or index, is to import the table, view, or synonym definition into the Clarion Data Dictionary. See *Import the Table Definitions*.

**Tip:** Defining a key as case insensitive can degrade performance because the driver and templates must generate calls to UPPER() to ensure case insensitivity. This forces the SQL backend to ignore the index in favor of a table scan.

## ABC Template and ABC Library Optimizations

---

The ABC Application Templates provides some options to optimize BrowseBox performance for SQL back ends. See *Template Overview—Classes Tab Options—Global—BrowseClass Configuration* for more information on these options. See also *BrowseClass Properties—ActiveInvisible, AllowUnfilled, and RetainRow*.

**Tip:** For best performance, we strongly recommend using the ABC Templates to generate your SQL based applications.

The ABC Library provides some properties to optimize BrowseBox performance for SQL back ends. These properties control the automatic record set buffering for the underlying VIEW. The ABC Library automatically implements buffering for SQL based BrowseBoxes. See *ViewManager Properties—PagesAhead, PagesBehind, PageSize, and TimeOut* for more information on these options.

## Inner Joins

---

For ABC Template applications, if you know that the parent **MUST** have one or more children in a lookup situation, use the **File Schematic** dialog to mark the join as an INNER join. Note that this only helps ABC Template applications. See INNER in the *Language Reference*.

## Filter (Contracting) Locators

---

Using Filter Locators on your BrowseBox controls rather than Incremental or Step Locators can reduce the volume of data sent between client and server. See *BrowseBox Control Template* for more information on Filter Locators.

## Approximate Record Count

---

By default, the Clarion templates generate code to count the total number of records to be processed for a report. This total record count allows for an

accurate progress bar display during report generation. However, for large tables, the resulting `SELECT COUNT(*)` can be very slow.

Therefore, for large reports, we recommend providing an approximate record count to suppress the `SELECT COUNT(*)` as follows:

1. In the **Application Tree** dialog, RIGHT-CLICK the (*Report*) procedure, then choose **Properties** from the popup menu.

This opens the **Procedure Properties** dialog.

2. Press the **Report Properties** button to open the **Report Properties** dialog.
3. In the **Record Filter** field, type *1* to enable the **Approx. Record Count** field.

Another filter you can use in almost every case is `PrimaryKey > 0`. In some cases this can actually speed up your report by forcing access through the primary key.

4. In the **Approx. Record Count** field, type an approximate record count for the report, such as *5000*.
5. Press the **OK** button to close the **Report Properties** dialog.
6. Press the **OK** button again to return to the **Application Tree** dialog.
7. Press the  button to save your work.

## Fixed Thumbs and Movable Thumbs

---

By default, Clarion's code generation Wizards use Fixed Thumbs when Browsing SQL tables because Movable Thumbs can cause major performance slow downs on large tables in Clarion / SQL applications. For this reason, we recommend that you specify Fixed Thumbs for your manually place BrowseBox controls as follows:

1. In the **Application Tree** dialog, RIGHT-CLICK the Browse procedure, then choose **Extensions** from the popup menu.

This opens the **Extension and Control Templates** dialog.

2. In the list box, select *Browse on ...*, then press the **Scroll Bar Behavior** button.

This opens the **Scroll Bar Behavior** dialog.

3. In the **Scroll Bar Type** drop-down list, select *Fixed Thumb*, then press the **OK** button.

4. Press the **OK** button again to return to the **Application Tree** dialog.

5. Press the  button to save your work.

## Batch Transaction Processing

---

Most SQL databases operate in auto-commit mode. This means that any operation that updates a table (ADD, PUT, or DELETE) executes an implicit COMMIT. This can be very slow for a series (batch) of updates.

To optimize batch processes, surround any batch processing in a transaction frame (that is, with LOGOUT and COMMIT). The LOGOUT command prevents any subsequent implicit COMMITs until the transaction frame ends with either a COMMIT or a ROLLBACK. For example:

```

 LOGOUT(.1,OrderDetail) !Begin Transaction
 DO ErrHandler !always check for errors
 LOOP X# = 1 TO RECORDS(DetailQue) !Process stored detail records
 GET(DetailQue,X#) !Get one from the QUEUE
 DO ErrHandler !check for errors
 Det:Record = DetailQue !Assign to record buffer
 ADD(OrderDetail) !and add it to the file
 DO ErrHandler !check for errors
 END
 COMMIT !Terminate good transaction

ErrHandler ROUTINE !Error routine
IF NOT ERRORCODE() THEN EXIT. !Bail out if no error
ROLLBACK !Rollback the bad transaction
MESSAGE('Transaction Error - ' & ERROR())!Log the error
RETURN !and get out

```

You may want to issue intermittent calls to COMMIT and LOGOUT to save data at regular intervals. See the *Language Reference* for more information.

## Duplicated Records in BrowseBox

---

There are some circumstances that can cause a BrowseBox to display a record more than once even though the record exists only once in the database. If you see this effect, you can try the following things to correct it.

- (ABC Template)Add a filter to your BrowseBox—the driver automatically forces a unique sort on the assumption that a filtered browse is small, so the overhead for the server to build a temporary index should be low.
- (ABC Template)Disable the AllowUnfilled option (ABC Templates, Global Properties, Classes tab, Browser, Configure, AllowUnfilled).
- (ABC Template)Disable the RetainRow option (ABC Templates, Global Properties, Classes tab, Browser, Configure, RetainRow).
- (Clarion Template)Change your key definition to specify enough fields to create a unique collation sequence.

## Explanation

Assume 5 records containing

| ID | Name |
|----|------|
| 1  | B    |
| 2  | C    |
| 3  | C    |
| 4  | A    |
| 5  | B    |

Browse these in ID order from record 1. Then switch tabs (sort by name) and expect record 1 to be still highlighted (RetainRow). The system issues a `SELECT * FROM table WHERE Name >= 'B' ORDER BY Name`, which returns

|   |   |
|---|---|
| 1 | B |
| 5 | B |
| 2 | C |
| 3 | C |

Because the BrowseBox is only partially filled, the system tries to fill backwards (AllowUnfilled) from the highlighted row (RetainRow). The system issues a `SELECT * FROM table WHERE Name <= 'B' ORDER BY Name DESC` (no unique key). This might return

|   |   |
|---|---|
| 1 | B |
| 5 | B |
| 4 | A |

Notice that records 1 and 5 appear in the same order as before (not reversed as you might expect). So the BrowseBox displays:

|   |                   |
|---|-------------------|
| 4 | A                 |
| 5 | B !phantom record |
| 1 | B                 |
| 5 | B                 |
| 2 | C                 |
| 3 | C                 |

## Using Embedded SQL

You can use Clarion's property syntax (PROP:SQL) to send SQL statements to the backend SQL server, within the normal execution of your program. For backward compatibility, you can also use the SEND function to send SQL statements; however, we recommend using the property syntax.

### PROP:SQL

---

You can use Clarion's property syntax (PROP:SQL) to send SQL statements to the backend SQL server, within the normal execution of your program. You can send any SQL statements supported by the SQL server.

This capability lets your program do backend operations independent of the SQL Accelerator driver's generated SQL. For example, multi-record updates can often be accomplished more efficiently with a single SQL statement than with a template generated Process procedure which updates one record at a time. In cases like these it makes sense for you to take control and send custom SQL statements to the backend, and PROP:SQL lets you do this.

When an error is encountered on the SQL back-end, ERRORCODE() returns 90, and you can use FILEERRORCODE() and FILEERROR() to return the error code and error message set by the back-end SQL server.

You may also query the contents of PROP:SQL to get the last SQL statement issued by the file driver.

Examples:

```
SQLFile{PROP:SQL}='SELECT field1,field2 FROM table1' |
 & 'WHERE field1 > (SELECT max(field1)' |
 & 'FROM table2' !Returns a result set you
 ! get one row at a time
 ! using NEXT(SQLFile)
```

```
SQLFile{PROP:SQL}='CALL GetRowsBetween(2,8)'!Call stored procedure
```

```
SQLFile{PROP:SQL}='CREATE INDEX ON table1(field1 DESC)' !No result set
```

```
SQLFile{PROP:SQL}='GRANT SELECT ON mytable TO fred' !DBA tasks
```

```
SQLString=SQLFile{PROP:SQL} !Get last SQL statement
```

**Note:** When you issue a SELECT statement using PROP:SQL, the selected fields must match the fields declared in the named file or view.

For example:

```

MyFile FILE,DRIVER('ODBC')
Record RECORD
Field1 LONG
Field2 STRING(10)
 END
 END

MyView VIEW(MyFile)
 PROJECT(MyFile.Field1)
 END

CODE
OPEN(MyFile)
OPEN(MyView)
MyView{PROP:SQL} = 'SELECT COUNT(*) FROM MyTable' !correct
NEXT(MyView) !correct

MyFile{PROP:SQL} = 'SELECT COUNT(*) FROM MyTable' !illegal
NEXT(MyFile) !illegal

```

## SEND

---

You can use the Clarion SEND procedure to send an SQL command to the backend database. This is provided for backward compatibility with early versions of Clarion. We recommend using the property syntax to send SQL statements to the backend database.

Examples:

```

SEND(SQLFile,'SELECT field1,field2 FROM table1' |
 & 'WHERE field1 > (SELECT max(field1))' |
 & 'FROM table2') !Returns a result set you
 ! get one row at a time
 ! using NEXT(SQLFile)

SEND(SQLFile,'CALL GetRowsBetween(2,8)') !Call stored procedure

SEND(SQLFile,'CREATE INDEX ON table1(field1 DESC)') !No result set

```

## Using Embedded SQL for Batch Updates

---

SQL does a good job of handling batch processing procedures such as: printing reports, displaying a screen full of table rows, or updating a group of table rows.

The SQL Accelerator drivers take full advantage of this when browsing a table or printing. However, they do not always use it to its best advantage with the Process template or in code which loops through a table to update multiple records. Therefore, when doing batch updates to a table, it can be much more efficient to execute an embedded SQL statement than to rely on the code generated by the Process template.

For example, to use PROP:SQL to increase all Salesman salaries by 10% you could:

```
SQLFile FILE,DRIVER('Oracle'),NAME(SalaryFile)
Record RECORD
SalaryAmount PDECIMAL(5,2),NAME('JOB')
. . .
CODE
SqlFile{PROP:SQL} = 'UPDATE SalaryFile SET '&|
 'SALARY=SALARY * 1.1 WHERE JOB='S'''
```

The names used in the SQL statement are the SQL table names, not the Clarion field names.

## Calling a Stored Procedure

---

### CALL

To call a stored procedure you normally use property syntax to issue the standard SQL syntax 'CALL storedprocedure.' For example:

```
file{PROP:SQL} = 'CALL SelectRecordsProcedure'
```

### NORESULTCALL

The SQL Accelerator drivers also allow the syntax 'NORESULTCALL storedprocedure' for stored procedures that do not return a result set.

### Return Values

The Accelerator drivers do not support return codes from stored procedures. You can have your stored procedures return a result set instead. For example:

```
MyFile FILE,DRIVER('ODBC')
Record RECORD
ErrorCode LONG
ErrorMsg STRING(100)
. . .
CODE
OPEN(MyFile)
MyFile{PROP:SQL} = 'CALL ProcWithResultSet'
NEXT(MyFile)
IF ~ERRORCODE()
 IF MyFile.ErrorCode THEN STOP(MyFileErrorMsg).
END
```

**Note:** The result set must match the fields declared in the named file or view.

## Debugging Your SQL Application

All of TopSpeed's SQL Accelerator drivers can create a log file documenting Clarion I/O statements they process, the corresponding SQL statements, and the SQL return codes.

You can generate system-wide logs and on-demand logs (conditional logging based on your program logic).

**Tip:** The Trace example program (installed by default to `Clarion5\Examples\Resource\Trace`) manages the .INI settings for all database drivers, as well as the Clarion view engine.

### System-wide Logging

---

For system-wide logging, you can add the following to your WIN.INI file:

```
[CWdriver]
Profile=[1|0]
Details=[1|0]
Trace=[1|0]
TraceFile=[Pathname]
```

where *driver* is the database driver name (for example [CWORACLE]). Neither the INI section name [CW*driver*] nor the INI entry names are case sensitive.

Profile=1 tells the driver to include the Clarion I/O statements in the log file; Profile=0 tells the driver to omit Clarion I/O statements. The Profile switch must be turned on for the Details switch to have any effect.

Details=1 tells the driver to include record buffer contents in the log file; however, if the file is encrypted, you must turn on both the Details switch and the ALLOWDETAILS switch to log record buffer contents (see *ALLOWDETAILS*). Details=0 tells the driver to omit record buffer contents. The Profile switch must be turned on for the Details switch to have any effect.

Trace=1 tells the driver to include all calls to the back-end file system, including the generated SQL statements and their return codes, in the log file. Trace=0 omits these calls. The Trace switch generally generates log information that helps TopSpeed debug the SQL drivers, but is not particularly useful to the developer.

TraceFile names the log file to write to. If TraceFile is omitted the driver writes the log to *driver.log* in the current directory. *Pathname* is the full pathname or the filename of the log file to write. If no path is specified, the driver writes the specified file to the current directory.

Logging opens the named logfile for exclusive access. If the file exists, the new log data is appended to the file.

## On Demand Logging

---

For on-demand logging you can use property syntax within your program to conditionally turn various levels of logging on and off. The logging is effective for the target table and any view for which the target table is the primary table.

```
file{PROP:Profile}=Pathname !Turns Clarion I/O logging on
file{PROP:Profile}='' !Turns Clarion I/O logging off
PathName = file{PROP:Profile} !Queries the name of the log file
file{PROP:Log}=string !Writes the string to the log file
file{PROP:Details}=1 !Turns Record Buffer logging on
fFile{PROP:Details}=0 !Turns Record Buffer logging off
```

where *Pathname* is the full pathname or the filename of the log file to create. If you do not specify a path, the driver writes the log file to the current directory.

You can also accomplish on demand logging with a SEND() command and the LOGFILE driver string. See *LOGFILE* for more information.

## Language Level Error Checking

---

You can use the FILEERROR() and FILEERRORCODE() functions to capture messages and codes returned from the backend server to the SQL Accelerator driver. See the *Language Reference* for more information on these functions.

## SQL Driver Strings

There are switches or “driver strings” you can set to control the way your application creates, reads, and writes files with a specific driver. Driver strings are simply messages or parameters that are sent to the file driver at run-time to control its behavior. See *Common Driver Features—Driver Strings* for an overview of these runtime Database Driver switches and parameters.

**Tip:** A forward slash precedes all SQL driver strings. The slash allows the driver to distinguish between driver strings and SQL statements sent with the SEND function.

The SQL Accelerator Drivers support the following Driver Strings:

### ALLOWDETAILS

**DRIVER('SQLDriver', '/ALLOWDETAILS = TRUE | FALSE' )**

The ALLOWDETAILS driver string allows the SQL Accelerator driver to include record buffer contents in the log file for encrypted files.

The ALLOWDETAILS driver string works with the Details switch described in the *Debugging Your SQL Application* section.

### LOGFILE

**DRIVER('SQLDriver', '/LOGFILE [= Pathname] [[message]]' )  
[ LogFile" = ] SEND(file, '/LOGFILE [= Pathname] [[message]]' )**

The LOGFILE driver string turns logging on and off, and optionally writes a message to the log file. Turning the LOGFILE switch on writes Clarion I/O statements processed by the driver to the specified log file. The LOGFILE driver string is equivalent to the Profile switch described in the *Debugging Your SQL Application* section.

*Pathname* is the full pathname or the filename of the log file to write. If you do not specify a path, the driver writes the log file to the current directory. If Pathname is omitted, the driver writes the log to SQLDriver.log in the current directory.

If the log file already exists, the driver appends to it; otherwise, the driver creates the log file.

The *message* is optional, however, if included, it must be surrounded by square brackets ([]) and a space must precede the opening square bracket.

**Note:** /LOGFILE must be the last driver string specified by the DRIVER attribute.

## WHERE

[ Where" = ] SEND (file, '/WHERE [ where-clause ]')

The SQL Accelerator drivers automatically build SQL WHERE clauses when your Clarion code contains a SET followed by a NEXT or PREVIOUS. You can customize the driver generated WHERE clause by using the WHERE driver string.

The SEND must be executed after the SET statement and before the NEXT or PREVIOUS statement.

**Note:** The SET statement clears any WHERE clause set by the SEND statement.

Because the SQL driver's generated SELECT statement is not compiled until the NEXT or PREVIOUS statement, the SEND function posts no error code and returns no result. For example:

```
Ord FILE,PRE(Ord),DRIVER('ODBC'),NAME('ord')
NameDate KEY(+Ord:NameId,-Ord:Date)
Record RECORD
Name STRING(12),NAME('NameId')
Date DATE,NAME('OrderDate')
Type STRING(1),NAME('OrderType')
Details STRING(20),NAME('OrderDetails')

 .
CODE
Ord:Name = 'SMITH'
SET(Ord:NameDate,Ord:NameDate)
SEND(Orders, 'WHERE OrderType = 'M')
LOOP
 NEXT(Ord)
 !...some processing
END
```

This generates a SELECT statement similar to:

```
SELECT NameId,OrderDate,OrderType,OrderDetails FROM Ord
WHERE (NameID >= 'SMITH') AND (OrderType = 'M')
```

## SQL Driver Properties

You can use Clarion's property syntax to query and set certain SQL Accelerator driver properties. These properties are described below.

### PROP:Alias

---

PROP:Alias sets or returns the alias the SQL Accelerator driver uses when generating SELECT statements for a view. PROP:Alias only returns a value previously set using PROP:Alias. For example:

```
Customer{PROP:Alias} = 'C' !set new table alias
OldAlias" = Customer{PROP:Alias} = '' !use default alias
```

**Tip:** Prior to an assignment to PROP:Alias, the return value for PROP:Alias is undefined.

### PROP:ConnectionString

---

PROP:ConnectionString returns an SQL database's connection information. For example:

```
AFileOwner STRING(256)
AFile FILE,DRIVER('ODBC'),OWNER(AFileOwner)
CODE
AFileOwner='DataSource'
OPEN(AFile)
IF NOT ERRORCODE()
 AFileOwner=AFile{PROP:ConnectionString}
END
```

### PROP:Details

---

See the Details switch described in the *Debugging Your SQL Application* section.

### PROP:Disconnect

---

PROP:Disconnect CLOSEs any open files in the target file's database, then disconnects the application from the database.

## PROP:Inner

---

PROP:Inner is a writable property for SQL Accelerator drivers. This is useful for testing the ODBC USEINNERJOIN driver string. See *PROP:Inner* in the *Language Reference* for more information.

## PROP:Log

---

PROP:Log writes a string to the log file. For example:

```
AFile FILE,DRIVER('ODBC'),OWNER('DataSource')
CODE
OPEN(Afile)
IF NOT ERRORCODE()
 Afile{PROP:Log}='Afile opened:'&CLOCK()
END
```

## PROP:LogFile

---

Same as PROP:Profile—for backward compatibility.

## PROP:OrderAllTables

---

Setting PROP:OrderAllTables to True forces the SQL Accelerator driver to use linking fields and secondary files' key component fields, as well as the primary file's key component fields, in the ORDER BY clause it sends to the server. You may need this switch if you are using a Clarion VIEW that joins multiple tables. By default (View{PROP:OrderAllTables}=FALSE), the SQL Accelerator driver includes only the primary file's key components in the ORDER BY clause it sends to the SQL server. For example:

```
BRW1::View:Browse VIEW(Customer)
 PROJECT(CUST:CustNo)
 PROJECT(CUST:Name)
 PROJECT(CUST:Zip)
 PROJECT(CUST:CustNo)
 JOIN(ORD:ByCustomer,CUST:CustNo)
 PROJECT(ORD:OrderNo)
 PROJECT(ORD:OrderDate)
 END
 END
CODE
?BRW1::View:Browse{PROP:OrderAllTables} = TRUE
```

Accessing this VIEW then generates a SELECT statement similar to:

```
SELECT CustNo,Name,Zip,OrderNo,OrderDate FROM Customer,Ord
WHERE (Customer.CustNo = Ord.CustNo)
ORDER BY CustNo,OrderNo
```

## PROP:Profile

---

Setting PROP:Profile to true tells the driver to include Clarion I/O statements in the log file. See the Profile switch described in the *Debugging Your SQL Application* section.

Profile=1 tells the driver to include the Clarion I/O statements in the log file; Profile=0 tells the driver to omit Clarion I/O statements. The Profile switch must be turned on for the Details switch to have any effect.

Details=1 tells the driver to include record buffer contents in the log file; however, if the file is encrypted, you must turn on both the Details switch and the /ALLOWDETAILS switch to log record buffer contents (see *ALLOWDETAILS*). Details=0 tells the driver to omit record buffer contents. The Profile switch must be turned on for the Details switch to have any effect.

**Note:** /ALLOWDETAILS is only valid as a parameter of the DRIVER attribute (Driver Options field in the File Properties dialog). It is not valid with the SEND command.

## PROP:SQL

---

See *Using Embedded SQL*.

## PROP:SQLFilter

---

You can use PROP:SQLFilter to filter your VIEWS using native SQL code rather than Clarion code.

When you use PROP:SQLFilter, the SQL filter is passed directly to the server. As such it cannot contain the names of tables, variables, or functions that the server is not aware of; that is the filter expression must be valid SQL syntax with valid SQL table and column names. For example:

```
View{PROP:SQLFilter} = 'Date = TO_DATE('01-MAY-1996','DD-MON-YYYY')'
```

or

```
View{PROP:SQLFilter} = 'StrField LIKE 'AD%'''
```

Note that the SQL Accelerator incorporates the PROP:SQLFilter expression into the WHERE clause of a generated SELECT statement. The generated SELECT statement may reference one or more tables by aliases. If your filter also references tables (e.g., Customer.Name < 'T'), you must use the same alias names generated by the SQL Accelerator. By default, the SQL Accelerator uses the next letter of the alphabet as the alias name. For

example, the Accelerator uses 'A' as the alias for the first table in the generated SELECT statement, then 'B' for the next table, and so on. You can use PROP:Alias to control the alias names generated by the SQL Accelerator. See *PROP:Alias* for more information.

### **Combining VIEW Filters and SQL Filters**

When you use PROP:SQLFilter, the SQL filter may replace any filter specified for the VIEW, or it may be in addition to a filter specified for the VIEW. Prefix the SQL filter with a plus sign (+) to append the SQL filter to the existing VIEW filter. For example:

```
View{PROP:SQLFilter} = '+ StrField LIKE ''AD%'''
```

When you append the SQL filter by using the plus sign, the logical end result of the filtering process is (View Filter) AND (SQL Filter).

Omit the plus sign (+) to replace the Clarion filter with the SQL filter. When you replace the Clarion filter with the SQL filter by omitting the plus sign, the logical end result of the filtering process is simply (SQL Filter).

See *PROP:Filter* in the *Language Reference* for more information.

## **PROP:SQLJoinExpression**

You can use PROP:SQLJoinExpression to structure your VIEWS using native SQL code rather than Clarion code.

**Note:** Using PROP:SQLJoinExpression may hurt performance in some circumstances.

When you use PROP:SQLJoinExpression, the SQL join expression is passed directly to the server. As such it cannot contain the name of variables or functions that the server is not aware of; that is the join expression must be valid SQL syntax with valid SQL column names. For example:

```
View{PROP:SQLJoinExpression} = 'TO_DATE - FROM_DATE'
```

### **Combining VIEW Orders and SQL Orders**

When you use PROP:SQLJoinExpression, the SQL join expression may replace any the join specified for the VIEW, or it may be in addition to the join specified for the VIEW. Prefix the SQL join with a plus sign (+) to concatenate the SQL join expression to the existing VIEW join expression. For example:

```
View{PROP:SQLOrder} = '+ TO_DATE - FROM_DATE'
```

When you concatenate the SQL join by using the plus sign, the result set contains first the Clarion joined values, then the SQL joined values.

Omit the plus sign (+) to replace the Clarion join expression with the SQL join expression.

See *PROP:JoinExpression* in the *Language Reference* for more information.

**Tip:** **PROP:SQLJoinExpression only affects the JOIN portions of the VIEW declaration; it does not affect the PROJECT portions.**

## PROP:SQLOrder

---

You can use PROP:SQLOrder to sort your VIEWS using native SQL code rather than Clarion code.

**Note:** **Using PROP:SQLOrder may hurt performance in some circumstances.**

When you use PROP:SQLOrder, the SQL order is passed directly to the server. As such it cannot contain the name of tables, variables, or functions that the server is not aware of; that is the order expression must be valid SQL syntax with valid SQL column names. For example:

```
View{PROP:SQLOrder} = 'TO_DATE - FROM_DATE'
```

Note that the SQL Accelerator incorporates the PROP:SQLOrder expression into the ORDERBY clause of a generated SELECT statement. The generated SELECT statement may reference one or more tables by aliases. If your order expression also references tables (e.g., *Customer.Name < 'T'*), you must use the same alias names generated by the SQL Accelerator. By default, the SQL Accelerator uses the next letter of the alphabet as the alias name. For example, the Accelerator uses 'A' as the alias for the first table in the generated SELECT statement, then 'B' for the next table, and so on. You can use PROP:Alias to control the alias names generated by the SQL Accelerator. See *PROP:Alias* for more information.

### Combining VIEW Orders and SQL Orders

When you use PROP:SQLOrder, the SQL order may replace any order specified for the VIEW, or it may be in addition to the order specified for the VIEW. Prefix the SQL order with a plus sign (+) to append the SQL order to the existing VIEW order. For example:

```
View{PROP:SQLOrder} = '+ TO_DATE - FROM_DATE'
```

When you append the SQL order by using the plus sign, the result set is ordered first by the Clarion order expression, then by the SQL order expression.

Omit the plus sign (+) to replace the Clarion order with the SQL order.

See *PROP:Order* in the *Language Reference* for more information.



# 35 - AS400 ACCELERATOR

## Overview

### AS400 Server

---

See your IBM Documentation for complete information on the AS/400 Processor and its operating system.

### AS400 Accelerator

---

TopSpeed's AS400 Accelerator lets Clarion programs (PC based programs) access data stored on an IBM AS/400.

The AS400 Accelerator file driver automatically imports the AS/400 file definitions into the Clarion development environment, then lets you use standard Clarion file I/O statements as well as embedded SQL statements to access the AS/400 files.

The AS400 Accelerator is one of several TopSpeed SQL Accelerator Drivers for Clarion development tools. These SQL Accelerator Drivers share a common code base and many common features such as TopSpeed's unique, high speed buffering technology (see *BUFFER* in the *Language Reference*), common driver strings, and SQL logging capability. However, their primary purpose is to translate Clarion file commands into appropriate, efficient SQL statements specific to their respective SQL servers, and to handle any result sets returned by those servers.

AS400 Accelerator converts standard Clarion file I/O statements and function calls into optimized SQL statements, which it sends to the AS/400 server for processing. This means you can use the same Clarion code to access AS/400 tables as you do to access other file systems such as TopSpeed or Btrieve. It also means you can use Clarion template generated code with your AS/400 databases.

In addition to the automatically generated SQL statements, the AS400 Accelerator forwards any SQL statements you specify to the AS/400 server. The AS400 Accelerator interprets the result set returned from the AS/400 server and makes it available to your application program with the Clarion NEXT or PREVIOUS statement.

## Terminology

---

As you might expect for two different systems, the terminology for PC and AS/400 systems is somewhat different. For example, a “library” on the AS/400 contains one or more database files whereas a “library” on the PC contains executable code. In cases like these, this book makes every effort to describe the item in terms clearly understood by both audiences.

## Installation

---

❑ *To install the AS400 Accelerator:*

1. Insert the AS400 Accelerator disk1 into your PC's floppy drive.
2. Run **A:SETUP** where A: is the drive letter of your floppy drive.

Follow the instructions on your screen. You should install to the directory that already contains Clarion.

When everything is installed, the setup program optionally opens the AS400 Accelerator on-line help file. This file contains late breaking information about the AS400 Accelerator file driver.

3. Please read the late breaking information.

When you have finished reading, close the file.

4. Register the AS400 Accelerator driver with the Clarion development environment.
5. Add the Client Access/400 (CA/400) SHARED folder to your system PATH.

By default, the pathname is C:\Program Files\IBM\Client Access\Shared; however, if you install CA/400 elsewhere, you should use the actual pathname to which you installed. Typically, you set your System Path with a SETPATH command issued within your AUTOEXEC.BAT file.

## Registering the AS400 Accelerator

---

You must register the AS400 Accelerator with the Clarion development environment before you can use it.

❑ *To register the AS/400 file driver in Clarion:*

1. Start Clarion (DOUBLE-CLICK on the icon).
2. Choose **Setup ► Database Driver Registry**.
3. Press the **Add** button.

4. Highlight C4AS4.DLL (located in the ..\Clarion4\BIN directory) in the list box, then press the **OK** button.

This registers the AS400 Accelerator driver (both 16-bit and 32-bit versions).

5. Press the **OK** button.

## ***If You're New to Clarion...***

Developing applications with Clarion is generally the same process, regardless of the file system you choose. Therefore, we strongly recommend that you work through the tutorials in the *Getting Started* and *Learning Clarion* books to increase your understanding of the Clarion environment generally. Then work through the tutorial in this book to focus on the special considerations that apply when you develop applications with AS/400.

Clarion has the capability (through its templates) to automatically generate lots of code for browsing, updating, and reporting your AS/400 databases. The template-generated code is designed to work with a variety of file systems, including AS/400; however there are a few specific exceptions you should be aware of. These specific exceptions and their work-arounds are illustrated in the tutorial in this book.

In addition to working through the tutorials, please take a few moments to review the following topics:

- ◆ *Introduction*
  - System Requirements*
  - Registering the AS400 Accelerator*
- ◆ *Using AS400 Accelerator*
  - Overview*
- ◆ *SQL Accelerator Drivers in the Application Handbook*

## ***If You're New to the AS/400...***

The AS/400 is a computer with the database built into the hardware. On top of this is an operating system (OS/400) that is aware of the database.

OS/400 has a file system similar to DOS with files and directories. AS/400 files are called either physical files, logical files or tables. AS/400 directories are called either libraries or collections. The AS/400 has no concept of sub-directories. This book always refers to AS/400 files as files and AS/400 directories as libraries.

On top of OS/400 are a variety of programming languages for manipulating the operating system and the underlying database. One of the languages supported by the AS/400 is SQL (Structured Query Language). The particular version of SQL supported is DB2 for OS/400. The AS400 Accelerator uses DB2 statements to access your data on the AS/400. It is not necessary to know DB2 to use the driver; however some understanding of DB2 will allow you to get better performance from your application.

Some useful manuals from IBM on DB2 for OS/400 are:

- ◆ *OS/400 DB2/400 Database - An Overview (Reference SC41-3700-00)*
- ◆ *OS/400 DB2 for OS/400 Database Programming (Reference SC41-3701-01)*
- ◆ *DB2 for OS/400 SQL Reference (Reference SC41-3612-01)*

You may issue DB2 database management commands through the AS400 Accelerator driver to perform DBA tasks; however we do not recommend using Clarion data management statements like CREATE() to manipulate your AS/400 files. We recommend you use the native DB2 statements instead.

Developing applications with Clarion is generally the same process, regardless of the file system you choose, but there are some exceptions. We strongly recommend that you work through the tutorial in this book to focus on the special considerations that apply when you develop applications with AS/400 files.

In addition to working through the tutorial, please take a few moments to review the following topics:

- ◆ *Introduction*
  - System Requirements*
  - Registering the AS400 Accelerator*
- ◆ *Using AS400 Accelerator*
  - Overview*
- ◆ *Specifications*
- ◆ *SQL Accelerator Drivers in the Application Handbook*

## System Requirements

The AS400 Accelerator installer automatically installs all components required to begin development.

### Hardware

---

You can run the Clarion development environment, including the AS400 Accelerator driver, on any PC that meets the minimum system requirements for Microsoft Windows 3.x, Windows 95™, or Windows NT 3.51.

- ◆ Windows 3.x, minimum of 4 Megabytes of RAM recommended.
- ◆ Windows 95, minimum of 8 Megabytes of RAM recommended.
- ◆ Windows NT 3.51, minimum of 12 Megabytes of RAM recommended.
- ◆ All, minimum of 5 Megabytes free hard disk space.

The applications that you develop with Clarion will execute comfortably on PCs that meet only the minimum requirements for these operating systems.

Both the Clarion development environment and the applications that you develop will need network access to an operating IBM AS/400.

### Software—Developer Requirements

---

#### Windows 3.x

|        |                                                                                            |
|--------|--------------------------------------------------------------------------------------------|
| AS/400 | OS/400 v3.1 or higher.                                                                     |
| PC     | Clarion 4 or higher.<br>Client Access /400 v3r1 or higher.<br>MS ODBC Engine. <sup>1</sup> |

#### Windows 95

|        |                                                                                                                         |
|--------|-------------------------------------------------------------------------------------------------------------------------|
| AS/400 | OS/400 v3.1 or higher.                                                                                                  |
| PC     | Clarion 4 or higher.<br>Client Access /400 v3r1 or higher. <sup>2</sup><br>MS ODBC Engine v2.10 or higher. <sup>1</sup> |

Windows NT

|        |                                                                                                                         |
|--------|-------------------------------------------------------------------------------------------------------------------------|
| AS/400 | OS/400 v3.1 or higher.                                                                                                  |
| PC     | Clarion 4 or higher.<br>Client Access /400 v3r1 or higher. <sup>2</sup><br>MS ODBC Engine v2.10 or higher. <sup>1</sup> |

- 1 Installed by AS400 Accelerator installer.
- 2 Enable **16-bit Client Access support** on the **Other** tab of the **Client Access Properties** dialog. See *Software—Client Access/400 Configuration* for more information.

**Software—End User Requirements**

---

Windows 3.x

|        |                                                                   |
|--------|-------------------------------------------------------------------|
| AS/400 | OS/400 v3.1 or higher.                                            |
| PC     | Client Access /400 v3r1 or later.<br>MS ODBC Engine. <sup>1</sup> |

Windows 95—16-bit

|        |                                                                                                 |
|--------|-------------------------------------------------------------------------------------------------|
| AS/400 | OS/400 v3.1 or higher.                                                                          |
| PC     | Client Access /400 v3r1 or higher. <sup>2</sup><br>MS ODBC Engine v2.10 or higher. <sup>1</sup> |

Windows 95—32-bit

|        |                                                                                    |
|--------|------------------------------------------------------------------------------------|
| AS/400 | OS/400 v3.1 or higher.                                                             |
| PC     | Client Access /400 v3r1 or higher.<br>MS ODBC Engine v2.10 or higher. <sup>1</sup> |

Windows NT—16-bit

|        |                                                                                                 |
|--------|-------------------------------------------------------------------------------------------------|
| AS/400 | OS/400 v3.1 or higher.                                                                          |
| PC     | Client Access /400 v3r1 or higher. <sup>2</sup><br>MS ODBC Engine v2.10 or higher. <sup>1</sup> |

Windows NT—32-bit

|        |                        |
|--------|------------------------|
| AS/400 | OS/400 v3.1 or higher. |
|--------|------------------------|

PC Client Access /400 v3r1 or higher.  
MS ODBC Engine v2.10 or higher.<sup>1</sup>

- 1 Installed by the ODBC install program supplied with AS/400 Connectä. See *Advanced Topics—Distributing your AS/400 Applications*.
- 2 Enable **16-bit Client Access support** on the **Other** tab of the **Client Access Properties** dialog. See *Software—Client Access/400 Configuration* for more information.

## Software—Client Access/400 Configuration

---

Client Access/400 is highly configurable in order to support a variety of client/server environments. Generally, you should configure CA/400 Properties and CA/400 Connection Properties without regard to the AS400 Accelerator driver, because the CA/400 settings generally do not affect the operation of the driver.

However, there is one exception to this rule: when running 32-bit operating systems (Windows 95 or Windows NT) with 16-bit applications (including the Clarion development environment), you must enable **16-bit Client Access support** on the **Other** tab of the **Client Access Properties** dialog to allow the driver to do automatic EBCDIC to ANSI conversion. If this is not enabled, some data may not display correctly.

The **Stack provider** you select does not affect the AS400 Accelerator driver.

When you have installed and configured your CA/400 software, you should verify the configuration is correct before using the AS400 Accelerator driver. You can verify your CA/400 configuration by performing a simple data transfer between the AS/400 and your PC. See your CA/400 documentation for more information on performing the data transfer.

# Using AS400 Accelerator

## Importing AS/400 Files to a Data Dictionary

You can use the Import Wizard in Clarion's Dictionary Editor to generate AS/400 file, view, and alias declarations directly into a Clarion data dictionary (.DCT) file. This eliminates the need to manually define files.

The import feature imports one file, view, or alias at a time from an existing AS/400 database. It imports each field's name, data type, and size as well as any keys or indexes defined on the file.

- ❑ *To import AS/400 file declarations:*

### Connect to the AS/400 Database

1. From the Dictionary Editor, choose **File ► Import File**.

This opens the **Select File Driver** dialog where you can specify the file driver to use.

2. Select the *AS/400* file driver from the drop-down list, then press the **OK** button.

This starts the **AS/400 Table Import Wizard**.



3. In the **AS/400 Name** field, type the name of the AS/400 that contains the demo files.
4. In the **Username** field, type your AS/400 username.
5. In the **Password** field, type your AS/400 password.

The Wizard displays the password as asterisks.

## Filter the List of Tables to Import

- Optionally, provide a filter expression to limit the list of importable files, views, and aliases:

In the **Filter** field, type any valid SQL filter expression (or use no filter to see all importable objects). The Import Wizard imports objects listed in the QSYS2.SYSTABLES table, thus you may filter on the following fields in QSYS2.SYSTABLES:

TABLE\_NAME  
 TABLE\_OWNER  
 TABLE\_TYPE  
 COLUMN\_COUNT  
 ROW\_LENGTH  
 TABLE\_TEXT  
 LONG\_COMMENT  
 TABLE\_SCHEMA  
 LAST\_ALTERED\_TIMESTAMP  
 SYSTEM\_TABLE\_NAME  
 SYSTEM\_TABLE\_SCHEMA

For example:

```
TABLE_SCHEMA = 'MyLibrary'
```

or

```
TABLE_OWNER <> 'QSYS'
```

- Clear the **Disconnect from server when finished** box, then press the **Next** button.

Clearing the box maintains the connection so you can import several file definitions without reconnecting.

If you are not connected to the AS/400, the **AS/400 File Import Wizard** starts Client Access and establishes a connection.

The **AS/400 File Import Wizard** prompts you to select the AS/400 file, view, or alias to import.



## Import the File Definition

1. Highlight the file, view, or alias to import, then press the **Finish** button.

If the file you imported has no primary key, the **AS/400 File Import Wizard** reminds you that you must manually define a primary key for the file.



If no unique key is defined for the AS/400 file, but there is a set of fields that uniquely defines each record, then you can add a key definition into your Clarion dictionary that consists of these fields. The driver will then operate with full flexibility, but may perform slower than if there was a unique key defined on the AS/400. See the *Dictionary Editor* chapter in the *User's Guide* for more information on defining keys.

The File definition is created and the **File Properties** dialog appears allowing you to make any desired modifications.



Notice that the AS/400 host name, Username, and Password are entered by default in the **Owner Name** field. Supplying this information here lets applications based on this data dictionary access the AS/400 file without supplying Username and Password at runtime. This may be appropriate for some applications; however, if data security is a consideration, you may want to remove this information from the **Owner Name** field so that Username and Password are supplied by the end user at runtime.

Also, please notice that the fully qualified file name, including the library (e.g., TOPLIB.CLIENTF) is entered by default in the **Full Pathname** field. Supplying this information here lets applications based on this data dictionary access the correct AS/400 file when your Clarion code references it by Clarion NAME or PREFIX.

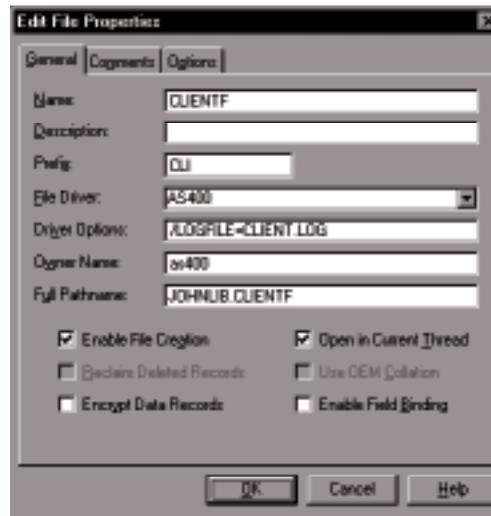
Import additional files by repeating these steps. After all the files are imported, return to the Dictionary Editor where you can define file relationships and delete any fields not used in your Clarion application. See *SQL Accelerator Drivers—Define Only the Fields You Use in the Application Handbook*.

## Table Names and Aliases

**Tip:** Generally, you should not have to manually specify table names. Rather, you simply import file definitions from your AS/400 database into your Clarion data dictionary. The AS400 Accelerator driver automatically uses the appropriate names (see *Importing AS/400 Files to a Data Dictionary*).

### NAME Attribute

The Clarion FILE statement's NAME attribute contains the AS/400 table name. The NAME attribute comes from the **Full Pathname** field in the Dictionary Editor's **File Properties** dialog.



In addition to the file name, you may specify a library name. If you omit the library name, the default library is used. For example, to access the *default.DBF* file on the AS/400, you create a Clarion file definition like:

```
DBFile FILE,DRIVER('AS400'),NAME('DBFILE')...
```

To specify a non-default library, use the fully qualified name as follows:

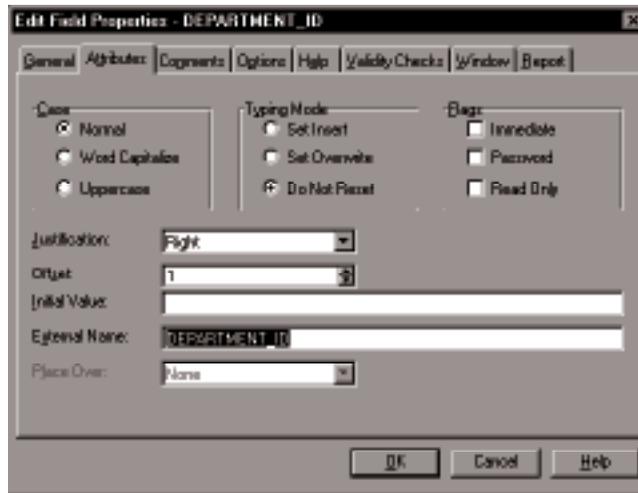
```
DBFile FILE,DRIVER('AS400'),NAME('LIBRARY.DBFILE')...
```

The NAME attribute may also contain the name of a view or alias on the AS/400.

## Column Names and Key Names

**Tip:** Generally, you should not have to manually specify column or key names. Rather, you simply import file definitions from your AS/400 database into your Clarion data dictionary. The AS400 Accelerator driver automatically uses the appropriate names (see *Importing AS/400 Files to a Data Dictionary*).

Since the AS/400 driver communicates with the AS/400 ODBC server using SQL statements, it must know how to reference the SQL columns and keys. The NAME attribute on each field or key of the Clarion FILE declaration contains this information. The NAME attribute comes from the **External Name** field in the **Field Properties** dialog or the **Key Properties** dialog.



```
Order FILE,PRE(Ord),DRIVER('AS400'),NAME('MyLib.Ord')
Date KEY(+Ord:Date),NAME('MyLib.DateKey') !from Key Properties
Record RECORD
Name STRING(12),NAME('NameId') !from Field Properties
Date DATE,NAME('OrderDate') !from Field Properties
END
END
```

**Note:** For keys you should specify a library name in which to create the key file. If you omit the library name, the AS/400 uses the default library.

An alternative way to supply the SQL column names is to add the BINDABLE attribute to the FILE definition. If the BINDABLE attribute is present, and there is no NAME attribute, the AS400 Accelerator driver uses the Clarion field label to reference the SQL columns. To add the BINDABLE attribute, in the Dictionary Editor, open the **File Properties** dialog and check the **Enable Field Binding** box (see the *Language Reference* for more information).

## Username and Passwords

### OWNER Attribute

The Clarion FILE statement's OWNER attribute optionally contains the AS/400 table's connection information (Username and Password). The OWNER attribute comes from the **Owner Name** field in the Dictionary Editor's **File Properties** dialog.

**Note:** For security reasons we recommend that you specify a variable connect string or no connect string in your Clarion FILE definition.

The syntax for OWNER is:

```
OWNER('AS/400 Name,[UserId],[Password][;DBQ=Lib1,Lib2,...,LibN]')
```

where DBQ specifies a list of AS/400 libraries in which to search for unqualified tables. That is, if the external name of a table is of the form *library.table*, then the table must be in library. If the external name is of the format *table*, then the table must exist in the DBQ list. If this list is not supplied then the table must reside in the QGPL library.

For example, to access the DBFILE file from the AS/400, you create a Clarion file definition like:

```
DBFile FILE,DRIVER('AS400'),NAME('DBFILE'). |
 OWNER('AS4001,Scott,tiger')
Record RECORD
ItemName STRING(5),NAME('ITEMNM')
Description STRING(25),NAME('ITEMD')
Quantity STRING(5),NAME('ITEMQ')
. . .
```

To store the connection string in a variable, rather than hard coding it into your dictionary, type a variable name in the **Owner Name** field in the Dictionary Editor's **File Properties** dialog, or create a Clarion file definition like this:

```
DBFile FILE,DRIVER('AS400'),NAME('DBFILE'),OWNER(Glo:LoginString)
Record RECORD
ItemName STRING(5),NAME('ITEMNM')
Description STRING(25),NAME('ITEMD')
Quantity STRING(5),NAME('ITEMQ')
```

. .

### **Accessing Multiple AS/400 Servers**

You can access tables from more than one AS/400 server simply by naming the different servers in the connect string for the table.

## Driver Behavior

AS400 Accelerator is one of several TopSpeed SQL Accelerator drivers. These SQL Drivers share a common code base and many common features such as TopSpeed's unique, high speed buffering technology, common driver strings, and SQL logging capability. **See *SQL Accelerators* for information on these common features.**

All the common behavior of all the SQL Accelerator drivers is documented in the *SQL Accelerators* chapter. All behavior specific to the AS400 Accelerator driver is noted in this chapter.

## Logging In

---

The AS400 Accelerator driver automatically looks for active Username and Password values whenever an AS/400 file is accessed. If no values have been supplied, the driver issues an error. If a Username and Password have already been supplied, the driver uses those values.



When all AS/400 files are closed, there are no active Username and Password values, so when another file is accessed, the driver needs this information again. Therefore, we recommend opening a file at the start of your program and keeping the file open throughout the session so your user only supplies the Username and Password one time.

Clarion's Application Wizard automatically generates code to do this for SQL based drivers such as AS400 Accelerator. However, if you do not use the Application Wizard, you can accomplish the same effect simply by adding an AS/400 file to the File Schematic for your main procedure. This automatically generates code to open the file, which then remains open as long as the main procedure is active.

## Performance Considerations

Generally, Clarion's development environment (Data Dictionary Import Wizard, Database Drivers, and templates) produce optimized, high performance, SQL applications.

**Tip:** For best performance, we strongly recommend using the ABC Templates to generate your SQL based applications.

The common performance considerations of the SQL Accelerator drivers is documented in the *SQL Accelerators* chapter. All performance considerations specific to the AS400 Accelerator are noted in this chapter.

## Ascending and Descending Keys

---

If you have a key on a field in ascending order and issue a

```
SET(KEY);PREVIOUS(File)
```

the AS/400 does not use the ascending key. Instead it creates a temporary key. To avoid this potential performance problem, create another AS/400 key with the same key component but the opposite order (descending). The new key need not be defined in your Clarion data dictionary.

## Driver Strings

There are switches or “driver strings” you can set to control the way your application creates, reads, and writes files with a specific Accelerator driver. Driver strings are simply messages or parameters that are sent to the file driver at run-time to control its behavior. **See *SQL Accelerators in the Application Handbook* for more information on SQL Accelerator driver strings.**

In addition to the common SQL Accelerator driver strings, AS400 Accelerator supports the following driver strings as well.

### USETRANSACTIONS

```
DRIVER('AS400', 'USETRANSACTIONS = TRUE | FALSE ')
['Transact' =] SEND(file, 'USETRANSACTIONS [= TRUE | FALSE]')
```

The fastest way to access the AS/400 is to not use transaction processing (LOGOUT, COMMIT, ROLLBACK). By default (USETRANSACTIONS=False), the AS400 Accelerator does not use transaction processing, that is, LOGOUT statements are ignored.

If your application requires transaction framing, then you must set USETRANSACTIONS=TRUE for all files included within a transaction frame. The first time LOGOUT is called with a file that has USETRANSACTIONS set to true, the AS/400 connection is changed to use transaction processing.

Once you enable transaction processing, it remains enabled until you drop the AS/400 connection (by closing all files on that AS/400).

**Note:** For a file to be accessed within a transaction it must have journaling active for it on the AS/400.

For a file to be accessed within a transaction it must have journaling active for it on the AS/400. For example, if you have a file CUSTOMER in the ACCOUNT library then the AS/400 commands to enable journaling are:

```
CRTJRNRCV ACCOUNT/QSQJRN
CRTJRN ACCOUNT/QSQJRN ACCOUNT/QSQJRN
STRTRJRNPF ACCOUNT/CUSTOMER ACCOUNT/QSQJRN
```

### GATHERATOPEN

```
DRIVER('AS400', 'GATHERATOPEN = TRUE | FALSE ')
```

By default the driver delays gathering field information until it is required. However, some backends (like Sybase 11) perform poorly under these conditions. Setting GATHERATOPEN to TRUE forces the driver to gather most of the field information when the file is opened, which avoids a slowdown during program execution.

## Driver Properties

You can use Clarion's property syntax to query and set certain AS400 Accelerator driver properties. In addition to the standard SQL Accelerator properties (see *SQL Accelerators—SQL Accelerator Properties*), the AS400 Accelerator supports the following properties.

### PROP:LogonScreen

---

PROP:LogonScreen sets or returns the toggle that determines whether the driver automatically prompts for logon information. By default (PROP:LogonScreen=True), the driver does display a logon window if no connect string is supplied. If set to False and there is no connect string, the OPEN(file) fails and FILEERRORCODE() returns '80001.' For example:

```
Afile FILE,DRIVER('AS400')
!file declaration with no userid and password
END
CODE
Afile{PROP:LogonScreen}=True !enable auto login screen
OPEN(Afile)
```

The automatic logon screen lets prompts for the following connection information. Consult your ODBC documentation for more information on these prompts:

#### Server

Select the workstation running the AS/400 database to import from. If the **Server** list is empty, you may type in the name. See your DBA or network administrator for information on how the server is specified.

#### Logon ID

For Standard Security, type your AS/400 Username. For Trusted Security (Integrated NT Security) no Username is required. See your server documentation or your DBA for information on applicable Usernames and security methods.

#### Password

For Standard Security, type your AS/400 Password. For Trusted Security (Integrated NT Security) no Password is required. See your server documentation or your DBA for information on applicable Passwords and security methods.

#### Database

Select the AS/400 database that contains the tables or views to access. If the **Database** list is empty, you may type in the name. See your server documentation or your DBA for information on database names.

# Specifications

## File Specification Maximums

---

The following are maximum values for AS/400 files accessed through Clarion products:

File Size : 266,757,734,400 bytes  
 Records per File : 2,147,483,646  
 Record Size: 32,766 bytes (32,740 if there is a VARCHAR)  
 Field Size : 32,766 bytes (32,740 if a VARCHAR)  
 Fields per Record : 8,000  
 Keys/Indexes per File : 120 fields  
 Key Size : Maximum 32 components or 10,000 bytes  
 SQL Statement Size: 31,744 characters  
 Open Files : Maximum 10 SQL SELECTs active at once

## Supported Data Types

---

The following table defines the Clarion data types you use to access the specified AS/400 data types:

**Tip:** Generally, you should not have to do any manual matching of data types. Rather, you simply import file definitions from your AS/400 database into your Clarion data dictionary. The AS400 Accelerator driver automatically selects the proper data types.

| <u>Clarion Data Type</u> | <u>AS/400 Data Type</u>                       |
|--------------------------|-----------------------------------------------|
| STRING                   | CHAR                                          |
| CSTRING                  | VARCHAR                                       |
| SREAL                    | REAL or SINGLE PRECISION                      |
| REAL                     | DOUBLE PRECISION                              |
| LONG                     | INTEGER                                       |
| SHORT                    | SMALLINT                                      |
| DATE                     | DATE, PACKED DECIMAL or DECIMAL <sup>1</sup>  |
| TIME                     | TIME, PACKED DECIMAL, or DECIMAL <sup>2</sup> |
| PDECIMAL                 | DECIMAL or PACKED DECIMAL <sup>3</sup>        |
| DECIMAL                  | NUMERIC or ZONED DECIMAL <sup>3</sup>         |
| STRING(8)                | TIMESTAMP <sup>4</sup>                        |
| GROUP,OVER(theString)    |                                               |
| DATE                     |                                               |
| TIME                     |                                               |
| END                      |                                               |

## Notes

- 1 Declaring a DATE or TIME field creates and accesses AS/400 DATE or TIME fields by default. It can also access dates in other formats by using an external name.

Dates may be stored on the AS/400 as a DATE or as a PACKED DECIMAL or a DECIMAL. For either type, use the Clarion DATE datatype. However, for DECIMAL or PACKED DECIMAL dates, you should specify the date format in the NAME attribute for the DATE field.

The syntax for specifying the format string is:

```
DateField DATE, NAME('field_name | format_string')
```

There must be a space on either side of the vertical bar separator.

The values for format\_string and each corresponding AS/400 data type the driver accesses are defined in the table below. When creating a file with a formatted DATE field, the driver creates a PDECIMAL field of the appropriate size.

The format string represents how the AS/400 numeric field is interpreted. For example format\_string yyymmdd indicates that dates are stored in an 8 digit numeric with the century in the left two digits, the year in the next two, the month in the next two and the day in the right most two digits.

| Value   | AS/400 Data Type               |
|---------|--------------------------------|
| yyymmdd | DECIMAL (8,0) or PDECIMAL(8,0) |
| yymmdd  | DECIMAL (6,0) or PDECIMAL(6,0) |
| mmdyyy  | DECIMAL (8,0) or PDECIMAL(8,0) |
| mmdyy   | DECIMAL (6,0) or PDECIMAL(6,0) |
| ddmmyyy | DECIMAL (8,0) or PDECIMAL(8,0) |
| ddmmy   | DECIMAL (6,0) or PDECIMAL(6,0) |

- 2 Normally when you define a TIME field you will be referencing an AS/400 TIME field. Some data files on AS/400s store times in PACKED DECIMAL or DECIMAL format.

To allow easy access and manipulation of these time fields, the driver supports a format string in the NAME attribute on a TIME field to describe the data stored.

The syntax for specifying the format string is:

```
TimeField TIME, NAME('field_name | format_string')
```

There must be a space on either side of the vertical bar separator.

The values for `format_string` and each corresponding AS/400 data type the driver accesses are defined in the table below. When creating a file with a formatted TIME field then driver will create a PDECIMAL field of the appropriate size.

The format string represents how the AS/400 numeric field is interpreted. For example, `format_string hhmm` indicates that times are stored in a four digit number with the hours in the left two digits and the minutes in the right two digits.

| Value  | AS/400 Data Type               |
|--------|--------------------------------|
| hhmmss | DECIMAL (6,0) or PDECIMAL(6,0) |
| hhmm   | DECIMAL (4,0) or PDECIMAL(4,0) |

- 3 Clarion can access DECIMAL or NUMERIC fields defined with greater than 15 digit precision, however the data is automatically rounded to 15 digit precision (the maximum available in Clarion), so you should not update these fields.
- 4 You can read from or write to files containing TIMESTAMP fields. However, the Clarion CREATE statement will not create a new file containing this data type.

## Supported Attributes and Procedures

### File Attributes Supported

|                                                          |                |
|----------------------------------------------------------|----------------|
| CREATE .....                                             | Y <sup>1</sup> |
| DRIVER( <i>filetype</i> [, <i>driver string</i> ]) ..... | Y              |
| NAME .....                                               | Y              |
| ENCRYPT .....                                            | N              |
| OWNER( <i>password</i> ) .....                           | Y              |
| RECLAIM .....                                            | N              |
| PRE( <i>prefix</i> ) .....                               | Y              |
| BINDABLE .....                                           | Y              |
| THREAD .....                                             | Y              |
| EXTERNAL( <i>member</i> ) .....                          | Y              |
| DLL( <i>flag</i> ) .....                                 | Y              |
| OEM .....                                                | N              |

### File Structures Supported

|              |   |
|--------------|---|
| INDEX .....  | Y |
| KEY .....    | Y |
| MEMO .....   | N |
| BLOB .....   | N |
| RECORD ..... | Y |

### Index, Key, Memo Attributes Supported

|                             |   |
|-----------------------------|---|
| BINARY .....                | N |
| DUP .....                   | Y |
| NOCASE .....                | N |
| OPT .....                   | N |
| PRIMARY .....               | Y |
| NAME .....                  | Y |
| Ascending Components .....  | Y |
| Descending Components ..... | Y |
| Mixed Components .....      | Y |

### Field Attributes Supported

|            |   |
|------------|---|
| DIM .....  | N |
| OVER ..... | Y |
| NAME ..... | Y |

### File Procedures Supported

|                                                 |                 |
|-------------------------------------------------|-----------------|
| BOF( <i>file</i> ) .....                        | N               |
| BUFFER( <i>file</i> ) .....                     | Y               |
| BUILD( <i>file</i> ) .....                      | Y               |
| BUILD( <i>key</i> ) .....                       | Y               |
| BUILD( <i>index</i> ) .....                     | Y <sup>3</sup>  |
| BUILD( <i>index, components</i> ) .....         | Y <sup>3</sup>  |
| BUILD( <i>index, components, filter</i> ) ..... | N               |
| BYTES( <i>file</i> ) .....                      | Y <sup>10</sup> |
| CLOSE( <i>file</i> ) .....                      | Y               |
| COPY( <i>file, new file</i> ) .....             | N               |
| CREATE( <i>file</i> ) .....                     | Y <sup>1</sup>  |
| DUPLICATE( <i>file</i> ) .....                  | Y               |
| DUPLICATE( <i>key</i> ) .....                   | Y               |
| EMPTY( <i>file</i> ) .....                      | Y               |
| EOF( <i>file</i> ) .....                        | N               |
| FLUSH( <i>file</i> ) .....                      | N               |
| LOCK( <i>file</i> ) .....                       | N               |
| NAME( <i>label</i> ) .....                      | Y               |
| OPEN( <i>file, access mode</i> ) .....          | Y               |

|                                         |                 |
|-----------------------------------------|-----------------|
| PACK( <i>file</i> ) .....               | N               |
| POINTER( <i>file</i> ) .....            | N               |
| POINTER( <i>key</i> ) .....             | N               |
| POSITION( <i>file</i> ) .....           | N               |
| POSITION( <i>key</i> ) .....            | Y <sup>11</sup> |
| RECORDS( <i>file</i> ) .....            | Y               |
| RECORDS( <i>key</i> ) .....             | Y <sup>12</sup> |
| REMOVE( <i>file</i> ) .....             | Y               |
| RENAME( <i>file, new file</i> ) .....   | N               |
| SEND( <i>file, message</i> ) .....      | Y               |
| SHARE( <i>file, access mode</i> ) ..... | Y               |
| STATUS( <i>file</i> ) .....             | Y               |
| STREAM( <i>file</i> ) .....             | N               |
| UNLOCK( <i>file</i> ) .....             | N               |

### Record Access Supported

|                                               |                |
|-----------------------------------------------|----------------|
| ADD( <i>file</i> ) .....                      | Y              |
| ADD( <i>file, length</i> ) .....              | N              |
| APPEND( <i>file</i> ) .....                   | Y <sup>4</sup> |
| APPEND( <i>file, length</i> ) .....           | N              |
| DELETE( <i>file</i> ) .....                   | Y              |
| GET( <i>file, key</i> ) .....                 | Y              |
| GET( <i>file, filepointer</i> ) .....         | Y <sup>5</sup> |
| GET( <i>file, filepointer, length</i> ) ..... | N              |
| GET( <i>key, keypointer</i> ) .....           | N              |
| HOLD( <i>file</i> ) .....                     | Y <sup>6</sup> |
| NEXT( <i>file</i> ) .....                     | Y              |
| NOMEMO( <i>file</i> ) .....                   | N              |
| PREVIOUS( <i>file</i> ) .....                 | Y <sup>7</sup> |
| PUT( <i>file</i> ) .....                      | Y              |
| PUT( <i>file, filepointer</i> ) .....         | N              |
| PUT( <i>file, filepointer, length</i> ) ..... | N              |
| RELEASE( <i>file</i> ) .....                  | N              |
| REGET( <i>file, string</i> ) .....            | N              |
| REGET( <i>key, string</i> ) .....             | Y <sup>8</sup> |
| RESET( <i>file, string</i> ) .....            | N              |
| RESET( <i>key, string</i> ) .....             | Y <sup>9</sup> |
| SET( <i>file</i> ) .....                      | Y              |
| SET( <i>file, key</i> ) .....                 | N              |
| SET( <i>file, filepointer</i> ) .....         | N              |
| SET( <i>key</i> ) .....                       | Y              |
| SET( <i>key, key</i> ) .....                  | Y              |
| SET( <i>key, keypointer</i> ) .....           | N              |
| SET( <i>key, key, keypointer</i> ) .....      | N              |
| SKIP( <i>file, count</i> ) .....              | Y              |
| WATCH( <i>file</i> ) .....                    | N              |

### Transaction Processing Supported<sup>13</sup>

|                                                 |   |
|-------------------------------------------------|---|
| LOGOUT( <i>timeout, file, ..., file</i> ) ..... | Y |
| COMMIT .....                                    | Y |
| ROLLBACK .....                                  | Y |

### Null Data Processing Supported

|                                  |                 |
|----------------------------------|-----------------|
| NULL( <i>field</i> ) .....       | Y               |
| SETNULL( <i>field</i> ) .....    | Y <sup>14</sup> |
| SETNONNULL( <i>field</i> ) ..... | Y               |

## Notes

---

- 1 You can read from or write to files containing `TIMESTAMP` fields. However, you cannot use `CREATE` to create a new file containing this data type.
 

When you `CREATE` files and keys on the AS/400, you should specify the library in which to create the file or key, otherwise, a default library is used. Specify the library in the `NAME` attribute for the file or key:

```
MyFile FILE,DRIVER('AS400'),NAME('MyLib.MyFile')
DateKey KEY(+Ord:Name,+Ord:Date),NAME('MyLib.DateKey')
 .
 .
 .
```
- 3 The `BUILD(dynamic index)` and `BUILD(index)` statements do not perform any disk action. They only initialize internal AS/400 driver structures to track key order access and allow `SELECT` statements to be built when you issue `SET(key)` or `SET(key,key)` statements referencing the *index*.
- 4 The `APPEND` statement behaves identically to the `ADD` statement, that is, keys *are* updated by the `APPEND` statement.
 

`APPEND` functions can use the bulk copy functionality of the AS/400. Each series of `APPEND` functions must be preceded by a statement assigning `PROP:AppendBuffer` to the file in order to initiate bulk copy. `PROP:AppendBuffer` specifies the number of rows to be bulk copied in each batch. For example:

```
SQLfile{PROP:AppendBuffer}='1000'.
```
- 5 The `GET(file, filepointer)` statement is unsupported for all values of *filepointer* except *filepointer* = 0. In this case, the record position is cleared and `ERRORCODE 35` is returned.
- 6 Apart from the holding records, the `HOLD` statement has another use. Normally, the driver will not reread the record when you execute a `RESET/NEXT` to the current record. Executing a `HOLD` statement before the `RESET/NEXT` forces the driver to reread the record from disk.
- 7 You can't execute a `PREVIOUS` after a `SET(file)` statement. You can only examine the *file* in a forward order.
- 8 The `REGET` statement only works if you have a unique key defined for the file
- 9 The `RESET(key,position)/NEXT(file)` statement sequence is optimized to retrieve the record from the driver's internal buffer if the code is resetting to the current record. To force the driver to reread the record from disk, execute a `HOLD` statement before the `RESET/NEXT` sequence. This optimization is not in effect within a transaction frame.

- 10 The BYTES(file) function returns the number of records in the file *or* the number of bytes in the last record accessed. Following an OPEN statement, the BYTES function returns the number of records in the *file*. After the file has been accessed by GET, NEXT, ADD, or PUT, the BYTES function returns the size of the last *record* accessed.
- 11 The POSITION(*key*) function returns (1 + size of the *key* components + the size of the components of the file's primary key). This formula is true even if the first unique key is the same *key* you are positioning on. If no primary key is defined, then the first unique key is considered the primary key.

If there is no unique key, POSITION(*key*) returns 1 + size of the *key* components. In this case RESET(*key*) will reposition to the first occurrence of the key value, since there is no way of uniquely identifying a record. Therefore, the RESET may position on a different record.

- 12 The RECORDS(*key*) function returns the number of UNIQUE occurrences of the first element of the *key*. This is the same as doing an SQL statement of:

```
SELECT COUNT (DISTINCT key_field1) FROM table
```

- 13 Transaction processing must first be enabled with the USETRANSACTIONS switch. See *Using AS400 Accelerator—Driver Strings* for more information.
- 14 SETNULL(field) clears the contents of the field.

## Troubleshooting

### AS400 is not in the File Driver Drop-down List

---

If the Select File Driver list in the Clarion data dictionary does not include AS400, then you have not registered the AS400 driver with the Clarion environment.



Please follow the steps described in *Register the Driver*.

### Could Not Access Key Information

---

The following message appears when you try to import an AS/400 file definition into the Clarion data dictionary.



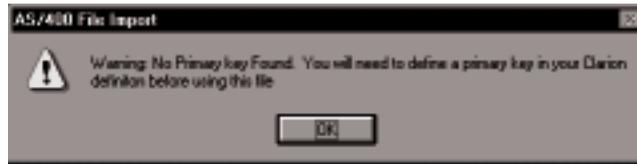
This message indicates the driver was unable to import keys for the imported file, typically because no keys are defined for the file on the AS/400.

You may modify the file on the AS/400 to include key information, or you may simply add the keys with the **Fields/Keys** button in the Clarion **Dictionary** dialog. See the *Dictionary Editor* chapter in the *User's Guide* for more information on defining keys.

### No Primary Key Found

---

The following message appears when you try to import an AS/400 file definition into the Clarion data dictionary.



This message indicates that although key information is available, there is no unique key or primary key for the imported file.

You may modify the file on the AS/400 to include a primary key, or you may simply use the **Fields/Keys** button in the Clarion **Dictionary** dialog to identify a unique or primary key for the file. See the *Dictionary Editor* chapter in the *User's Guide* for more information on defining keys.

## Unsupported Function

---

If your application issues an "Unsupported Function" message when you try to delete or change a record, you probably did not define a primary (or unique) key for your file.

You may modify the file on the AS/400 to include a primary key then reimport the file definition, or you may simply use the **Fields/Keys** button in the Clarion **Dictionary** dialog to identify a unique or primary key for the file. See the *Dictionary Editor* chapter in the *User's Guide* for more information on defining keys.

## Communication link failure

---

The following message appears when the AS400 Accelerator driver tries, but fails, to establish a connection to the AS/400.



This message usually indicates an invalid username or password.

## Windows 95

### Cannot Load the Driver

---

The following message appears when you try to import an AS/400 file definition into the Clarion data dictionary under Windows 95.



This message indicates the Microsoft ODBC engine is not installed in your Windows\System directory. The AS400 Accelerator install program should install this engine. Try reinstalling AS400 Accelerator, then confirm that ODBC.DLL, ODBCINST.DLL, and CTL3DV2.DLL are in your Windows\System directory.

### Could not Logon. Specified Driver Could not be Loaded

---

The following message appears when you try to import an AS/400 file definition into the Clarion data dictionary under Windows 95.



Under Windows 95, this message indicates you need to add the Client Access/400 SHARED folder to your system path.

A variation of this message may appear at runtime on the end user's machine if the Client Access/400 shared directory has not been added to the system path. See *System Requirements—Software—End User Requirements* for more information.

### CWBSY1004 Internal Error

---

The following message appears when you try to connect to the AS/400 using Client Access/400 for Windows 95.



Under Windows 95, this message indicates you have not entered a valid Client Access password. Client Access may require a password value even though no password is required by your AS/400 system.

## Example Application

The AS400 Accelerator example application is a simple order entry system created to demonstrate the AS400 Accelerator.

Before running the application, you must install AS400 Accelerator as described in the *Introduction*.

This section points out the programming techniques used to create the application. Although all procedures use the standard templates, some “tricks” have been used to optimize performance. In most cases these “tricks” are merely a line or two of embedded source code. In other cases the optimization is accomplished in the data dictionary. These methods are documented here.

The program also demonstrates some alternative methods to accomplish the same task. By understanding the different approaches, you can choose the best method for your program.

## Running the Example Program

---

After you start the program, you should begin by initializing the data files. Under the **Utilities** menu, choose the **Copy Data to Server** option or press the **Copy** button to create the AS/400 tables. This copies several small files to the AS/400 library you specify. When you are finished with the example program, you can use the **Remove Data Files** command to remove these tables from the AS/400. You can recreate the tables at any time by pressing the **Copy** button.

## Procedures

---

Most of the procedures in the application are standard template generated browses and forms. The following procedures are not standard template generated code, but are provided to illustrate various techniques for programming with Clarion and AS/400.

### LogInWindow

Procedure template: WINDOW

This procedure displays an introductory screen to collect AS/400 connection information. The connection information is stored in global variables where it is reused for each AS/400 table accessed.

Opening the first file takes longer than any other file because the initial connection to AS/400 is made at that time. By opening the first file from the first procedure, this time delay is incurred early on in the program.

### **BatchChange**

Procedure template: PROCESS

This is a standard Clarion batch process to update a field. Upon completion, another screen appears showing how long (in hundredths of a second) it took to do the update.

A faster method to accomplish the same task is demonstrated in the **SQLChange** procedure.

### **SQLChange**

Procedure template: SOURCE

This procedure does the same job as **BatchChange** except it uses SQL code instead of standard Clarion I/O functions. This improves performance. Upon completion, another screen appears showing how long (in hundredths of a second) it took to do the update.

### **SQLBuilder**

Procedure template: WINDOW

This procedure builds and executes an SQL select statement with PROP:SQL, then displays the results, demonstrating one way to provide ad hoc queries. Note however, that for the present, the number and data types of the SQL columns selected must match the number and data types of the fields declared in the receiving FILE structure. Therefore, you must declare a series of FILES that match the output of the potential SELECT statements, then you must control the number and sequence of columns selected, and finally, you must direct the output to the matching FILE.

### **SQLFreeForm**

Procedure template: WINDOW

This procedure builds and executes free form SQL code using PROP:SQL.

Use the pre-written SQL block to change a flag for all records in the Orders file, or write your own free form code. Do not execute code that returns data (that is a SELECT statement). Although this is legal (see the SQLBuilder procedure), this particular procedure is not designed to receive data.

## PrintClients

Procedure template: BROWSE

An intermediate procedure using a standard Browse to select a client for which to print invoices. The menu option calling this procedure dictates which Report procedure is called when the user selects a client.

There are two Report procedures, both of which print similar reports; however, different methods are used to generate each report. This demonstrates the flexibility of the Clarion and AS/400 tools, plus the performance trade-offs with each method.

**ClarionPrint** uses related files in the file schematic.

**ViewPrint** uses an existing view on the AS/400 server.

## ClarionPrint

Procedure template: REPORT

This report prints all Invoices for a single client using standard Clarion file definitions. This is the slower of the two report methods used.

## ViewPrint

Procedure template: REPORT

This report prints all Invoices for a single client using an AS/400 VIEW.

# 36 - MSSQL ACCELERATOR

## Overview

### MSSQL Server

---

For complete information on the MSSQL database system, please review Microsoft's SQL Server documentation.

### MSSQL Accelerator

---

The MSSQL Accelerator is one of several TopSpeed SQL Accelerator drivers. These SQL Drivers share a common code base and many common features such as TopSpeed's unique, high speed buffering technology, common driver strings, and SQL logging capability. **See *SQL Accelerators* for information on these common features.**

The MSSQL Accelerator converts standard Clarion file I/O statements and function calls into optimized SQL statements, which it sends to the backend MSSQL server for processing. This means you can use the same Clarion code to access both MSSQL tables and other file systems such as TopSpeed files. It also means you can use Clarion template generated code with your SQL databases.

All the common behavior of all the SQL Accelerators is documented in the *SQL Accelerators* chapter. All behavior specific to the MSSQL driver is noted in this chapter.

### SQL Import Wizard—Login Dialog

---

Clarion's Dictionary Editor Import Wizard lets you import MSSQL table definitions into your Clarion Data Dictionary. When you select the MSSQL Accelerator from the driver drop-down list, the Import Wizard opens the **Login/Connection** dialog. The **Login/Connection** dialog collects the connection information for the MSSQL database.

**Note:** If you are using a Trusted Connection (Integrated NT Security), you must establish a connection to the NT workstation running the MSSQL Server before you can connect to the MSSQL database and import table definitions. You can verify your connection by running the MSSQL ISQL\_w Server utility installed with your MSSQL Client software.

Fill in the following fields in the **Login/Connection** dialog:

### Servername

Select the workstation running the MSSQL database to import from. If the **Servername** list is empty, you may type in the name. See your DBA or network administrator for information on how the server is specified.



### Username

For Standard Security, type your MSSQL Username. For Trusted Security (Integrated NT Security) no Username is required. See your server documentation or your DBA for information on applicable Usernames and security methods.

### Password

For Standard Security, type your MSSQL Password. For Trusted Security (Integrated NT Security) no Password is required. See your server documentation or your DBA for information on applicable Passwords and security methods.

### Database

Select the MSSQL database that contains the tables or views to import. If the **Database** list is empty, you may type in the name. See your server documentation or your DBA for information on database names.

### Filter

Optionally, provide a filter expression to limit the list of tables and views to import. The filter expression queries the `dbo.sysobjects` table. The filter expression is limited to 1024 characters in length.

**Tip:** The filter is case sensitive, so type your filter value accordingly.

Following is a list of the column names (and their Clarion datatypes) you can reference in your filter expression. Generally,

filtering on MSSQL system tables requires not only an intimate knowledge of the MSSQL system tables, but also of the MSSQL stored procedures. For example, to filter on table owner:

```
uid = user_id('DB0')
```

See your SQL server documentation for information on the MSSQL system tables and stored procedures.

|            |             |
|------------|-------------|
| name       | CSTRING(31) |
| id         | LONG        |
| uid        | SHORT       |
| type       | STRING(2)   |
| userstat   | SHORT       |
| sysstat    | SHORT       |
| indexdel   | SHORT       |
| schema_ver | SHORT       |
| refdate    | STRING(8)   |
| crdate     | STRING(8)   |
| version    | LONG        |
| deltrig    | LONG        |
| instrig    | LONG        |
| updtrig    | LONG        |
| seltrig    | LONG        |
| category   | LONG        |
| cache      | SHORT       |

**Next >**

Press this button to open the Import Wizard's **Import List** dialog.

## SQL Import Wizard—Import List Dialog

When you press the **Next >** button, the Import Wizard opens the **Import List** dialog. The **Import List** dialog lists the importable items.



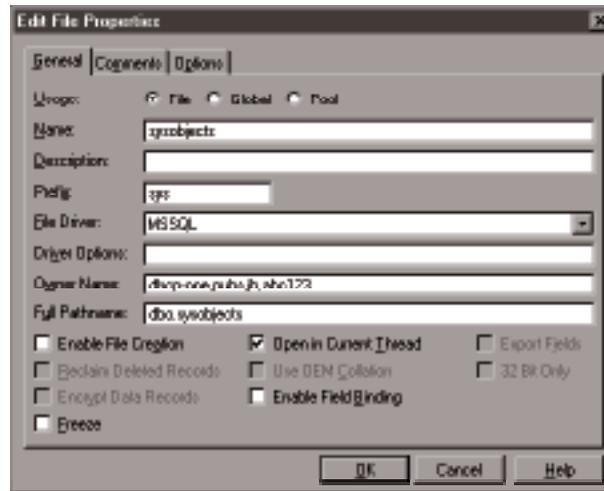
Highlight the table or view whose definition to import, then press the **Finish** button to import. The Import Wizard adds the definition to your Clarion Data Dictionary, then opens the **File Properties** dialog to let you modify the default definition.

Import additional tables or views by repeating these steps. After all the items are imported, return to the Dictionary Editor where you can define relationships and delete any columns not used in your Clarion application. See *SQL Accelerators—Define Only the Fields You Use*.

**Tip:** You can use the Clarion Enterprise Edition Dictionary Synchronizer to import an entire database, including table relationships, in a single pass.

## Connection Information and Driver Configuration—File Properties

Typically, you add MSSQL support to your application by importing the table definitions into your Clarion Data Dictionary. The Import Wizard automatically fills in the **File Properties** dialog with default values based on the imported item. However, you can use the **Owner Name** field in the **File Properties** dialog to further configure the way the MSSQL Accelerator accesses the data.



The OWNER attribute for MSSQL takes the format:

```
server,<database>,<uid>,<pwd><<LANGUAGE=language><<APP=name><<WSID=name>
```

**LANGUAGE**

The language used by MSSQL Server.

**APP**

The name of the application.

**WSID**

The workstation ID. Typically, this is the network name of the computer on which the application resides.

See your MSSQL Server documentation for information on these settings.

**Tip:** Type an exclamation point (!) followed by a variable name in the Owner Name field to specify a variable rather than hard coding the OWNER attribute . For example: !GLO:SQLOwner.

## ***Performance Considerations***

The MSSQL Accelerator uses cursors. The MSSQL Server will not use an Index with a cursor, but it will use a Unique Constraint with a cursor. Therefore we recommend using Unique Constraints rather than Indexes wherever possible.

## Driver Strings

There are switches or “driver strings” you can set to control the way your application creates, reads, and writes files with a specific driver. Driver strings are simply messages or parameters that are sent to the file driver at run-time to control its behavior. See *Common Driver Features—Driver Strings* for an overview of these runtime Database Driver switches and parameters.

**Tip:** A forward slash precedes all SQL Accelerator driver strings. The slash allows the driver to distinguish between driver strings and SQL statements sent with SEND.

In addition to the standard SQL Driver Strings, the MSSQL Accelerator supports the following Driver Strings:

### LOGONSCREEN

---

```
DRIVER('MSSQL', '/LOGONSCREEN = TRUE | FALSE ')
[AutoLogon" =] SEND(file, '/LOGONSCREEN [= TRUE | FALSE]')
```

See PROP:LogonScreen.

### GATHERATOPEN

---

```
DRIVER('MSSQL', '/GATHERATOPEN = TRUE | FALSE ')
```

By default the driver delays gathering field information until it is required. However, some backends (like Sybase 11) perform poorly under these conditions. Setting GATHERATOPEN to TRUE forces the driver to gather most of the field information when the file is opened, which avoids a slowdown during program execution.

### SAVESTOREDPROC

---

```
DRIVER('MSSQL', '/SAVESTOREDPROC= TRUE | FALSE ')
[SaveProc" =] SEND(file, '/SAVESTOREDPROC [= TRUE | FALSE]')
```

The MSSQL Accelerator executes SQL statements by creating temporary stored procedures on the server and executing them. By default (SAVESTOREDPROC=TRUE), these stored procedures remain on the server until connection to the server is dropped. To remove the procedures as soon as possible, set SAVESTOREDPROC=FALSE.

## TRUSTEDCONNECTION

---

```
DRIVER('MSSQL', '/TRUSTEDCONNECTION = TRUE | FALSE ')
[Trusted" =] SEND(file, '/TRUSTEDCONNECTION [= TRUE | FALSE]')
```

By default (TRUSTEDCONNECTION=FALSE), the MSSQL Accelerator requests a standard connection to SQL Server. To connect using SQL Server integrated security, set TRUSTEDCONNECTION=TRUE.

**Tip:** To set the connection type, you must issue the TRUSTEDCONNECTION switch before the connection is made to the server.

## Driver Properties

You can use Clarion's property syntax to query and set certain MSSQL Accelerator driver properties. In addition to the standard SQL Accelerator properties (see *SQL Accelerators—SQL Accelerator Properties*), the MSSQL Accelerator supports the following properties.

### PROP:LogonScreen

---

PROP:LogonScreen sets or returns the toggle that determines whether the driver automatically prompts for logon information. By default (PROP:LogonScreen=True), the driver does display a logon window if no connect string is supplied. If set to False and there is no connect string, the OPEN(file) fails and FILEERRORCODE() returns '28000.' For example:

```
Afile FILE,DRIVER('MSSQL')
!file declaration with no userid and password
END
CODE
Afile{PROP:LogonScreen}=True !enable auto login screen
OPEN(Afile)
```

The automatic logon screen lets prompts for the following connection information. Consult your MSSQL documentation for more information on these prompts:

#### Server

Select the workstation running the MSSQL database to import from. If the **Server** list is empty, you may type in the name. See your DBA or network administrator for information on how the server is specified.

#### Logon ID

For Standard Security, type your MSSQL Username. For Trusted Security (Integrated NT Security) no Username is required. See your server documentation or your DBA for information on applicable Usernames and security methods.

#### Password

For Standard Security, type your MSSQL Password. For Trusted Security (Integrated NT Security) no Password is required. See your server documentation or your DBA for information on applicable Passwords and security methods.

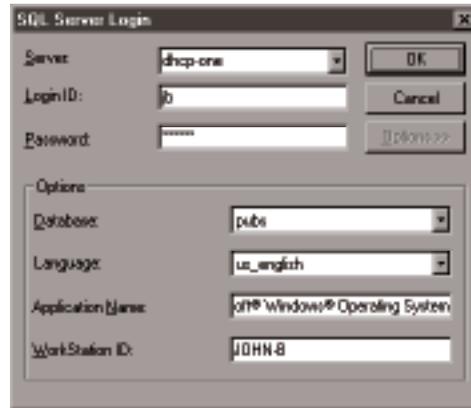
#### Options

Press this button to enable the following prompts. See your MSSQL Server documentation for information on these prompts.

## Supported Attributes and Procedures

### Database

Select the MSSQL database that contains the tables or views to access. If the **Database** list is empty, you may type in the name. See your server documentation or your DBA for information on database names.



### Language

The language used by MSSQL Server.

### Application Name

The name of the application.

### WorkStation ID

The workstation ID. Typically, this is the network name of the computer on which the application resides.

**File Attributes Supported**

|                                                          |                |
|----------------------------------------------------------|----------------|
| CREATE .....                                             | Y              |
| DRIVER( <i>filetype</i> [, <i>driver string</i> ]) ..... | Y              |
| NAME .....                                               | Y              |
| ENCRYPT .....                                            | N              |
| OWNER( <i>password</i> ) .....                           | Y <sup>1</sup> |
| RECLAIM .....                                            | N              |
| PRE( <i>prefix</i> ) .....                               | Y              |
| BINDABLE .....                                           | Y              |
| THREAD .....                                             | Y              |
| EXTERNAL( <i>member</i> ) .....                          | Y              |
| DLL( <i>flag</i> ) .....                                 | Y              |
| OEM .....                                                | N              |

**File Structures Supported**

|              |   |
|--------------|---|
| INDEX .....  | Y |
| KEY .....    | Y |
| MEMO .....   | N |
| BLOB .....   | N |
| RECORD ..... | Y |

**Index, Key, Memo Attributes Supported**

|                             |                |
|-----------------------------|----------------|
| BINARY .....                | N <sup>3</sup> |
| DUP .....                   | Y              |
| NOCASE .....                | Y              |
| OPT .....                   | N              |
| PRIMARY .....               | Y              |
| NAME .....                  | Y              |
| Ascending Components .....  | Y              |
| Descending Components ..... | Y              |
| Mixed Components .....      | Y              |

**Field Attributes Supported**

|            |   |
|------------|---|
| DIM .....  | N |
| OVER ..... | Y |
| NAME ..... | Y |

**File Procedures Supported**

|                                                 |                |
|-------------------------------------------------|----------------|
| BOF( <i>file</i> ) .....                        | N              |
| BUFFER( <i>file</i> ) .....                     | Y              |
| BUILD( <i>file</i> ) .....                      | Y              |
| BUILD( <i>key</i> ) .....                       | Y              |
| BUILD( <i>index</i> ) .....                     | Y <sup>3</sup> |
| BUILD( <i>index, components</i> ) .....         | Y <sup>3</sup> |
| BUILD( <i>index, components, filter</i> ) ..... | N              |
| BYTES( <i>file</i> ) .....                      | Y              |
| CLOSE( <i>file</i> ) .....                      | Y              |
| COPY( <i>file, new file</i> ) .....             | N              |
| CREATE( <i>file</i> ) .....                     | Y              |
| DUPLICATE( <i>file</i> ) .....                  | Y              |
| DUPLICATE( <i>key</i> ) .....                   | Y              |
| EMPTY( <i>file</i> ) .....                      | Y              |
| EOF( <i>file</i> ) .....                        | N              |
| FLUSH( <i>file</i> ) .....                      | N              |
| LOCK( <i>file</i> ) .....                       | N              |
| NAME( <i>label</i> ) .....                      | Y              |
| OPEN( <i>file, access mode</i> ) .....          | Y              |
| PACK( <i>file</i> ) .....                       | N              |
| POINTER( <i>file</i> ) .....                    | N              |

|                                         |   |
|-----------------------------------------|---|
| POINTER( <i>key</i> ) .....             | N |
| POSITION( <i>file</i> ) .....           | N |
| POSITION( <i>key</i> ) .....            | Y |
| RECORDS( <i>file</i> ) .....            | Y |
| RECORDS( <i>key</i> ) .....             | Y |
| REMOVE( <i>file</i> ) .....             | Y |
| RENAME( <i>file, new file</i> ) .....   | N |
| SEND( <i>file, message</i> ) .....      | Y |
| SHARE( <i>file, access mode</i> ) ..... | Y |
| STATUS( <i>file</i> ) .....             | Y |
| STREAM( <i>file</i> ) .....             | N |
| UNLOCK( <i>file</i> ) .....             | N |

**Record Access Supported**

|                                               |   |
|-----------------------------------------------|---|
| ADD( <i>file</i> ) .....                      | Y |
| ADD( <i>file, length</i> ) .....              | N |
| APPEND( <i>file</i> ) .....                   | Y |
| APPEND( <i>file, length</i> ) .....           | N |
| DELETE( <i>file</i> ) .....                   | Y |
| GET( <i>file, key</i> ) .....                 | Y |
| GET( <i>file, filepointer</i> ) .....         | N |
| GET( <i>file, filepointer, length</i> ) ..... | N |
| GET( <i>key, keypointer</i> ) .....           | N |
| HOLD( <i>file</i> ) .....                     | N |
| NEXT( <i>file</i> ) .....                     | Y |
| NOMEMO( <i>file</i> ) .....                   | N |
| PREVIOUS( <i>file</i> ) .....                 | Y |
| PUT( <i>file</i> ) .....                      | Y |
| PUT( <i>file, filepointer</i> ) .....         | N |
| PUT( <i>file, filepointer, length</i> ) ..... | N |
| RELEASE( <i>file</i> ) .....                  | N |
| REGET( <i>file, string</i> ) .....            | N |
| REGET( <i>key, string</i> ) .....             | Y |
| RESET( <i>file, string</i> ) .....            | N |
| RESET( <i>key, string</i> ) .....             | Y |
| SET( <i>file</i> ) .....                      | Y |
| SET( <i>file, key</i> ) .....                 | N |
| SET( <i>file, filepointer</i> ) .....         | N |
| SET( <i>key</i> ) .....                       | Y |
| SET( <i>key, key</i> ) .....                  | Y |
| SET( <i>key, keypointer</i> ) .....           | N |
| SET( <i>key, key, filepointer</i> ) .....     | N |
| SKIP( <i>file, count</i> ) .....              | Y |
| WATCH( <i>file</i> ) .....                    | Y |

**Transaction Processing Supported<sup>2</sup>**

|                                                 |                |
|-------------------------------------------------|----------------|
| LOGOUT( <i>timeout, file, ..., file</i> ) ..... | Y <sup>4</sup> |
| COMMIT .....                                    | Y              |
| ROLLBACK .....                                  | Y              |

**Null Data Processing Supported**

|                                  |   |
|----------------------------------|---|
| NULL( <i>field</i> ) .....       | Y |
| SETNULL( <i>field</i> ) .....    | Y |
| SETNONNULL( <i>field</i> ) ..... | Y |

## Notes

---

- 1 We recommend using a variable password that is lengthy and contains special characters because this more effectively hides the password value from anyone looking for it. For example, a password like “dd....#\$...\*&” is much more difficult to “find” than a password like “SALARY.”

**Tip:** To specify a variable instead of the actual password in the Owner Name field of the File Properties dialog, type an exclamation point (!) followed by the variable name. For example: !MyPassword.

- 2 See also *PROP:Logout* in the *Language Reference*.
- 3 BUILD(index) sets internal driver flags to guarantee the driver generates the correct ORDER BY clause. The driver does not call the backend server.
- 4 Whether LOGOUT also LOCKs the table depends on the server's configuration for transaction processing. See your server documentation.

## Synchronizer Server

Clarion's Enterprise Edition includes the MSSQL Synchronizer Server and the Data Dictionary Synchronizer. The Dictionary Synchronizer uses the Synchronizer Server to gather complete information about an MSSQL database.

The MSSQL Synchronizer Server is one of several used by the Dictionary Synchronizer. All the common behavior of the Synchronizer Servers is documented in the *Enterprise Tools* book. All behavior specific to the MSSQL Synchronizer Server is noted in this section.

### Synchronizer Login Dialog

---

Clarion's Dictionary Synchronizer Wizard (Enterprise Edition) lets you import an entire MSSQL database definition into your Clarion Data Dictionary in a single pass. During this process, the Synchronizer Wizard opens an MSSQL login dialog. This dialog collects the connection information for the MSSQL database.

**Note:** If you are using a Trusted Connection (Integrated NT Security), you must establish a connection to the NT workstation running the MSSQL Server before you can connect to the MSSQL database and import table definitions. You can verify your connection by running the MSSQL ISQL\_w Server utility installed with your MSSQL Client software.

Fill in the following fields in the login dialog:

#### Host

Select the workstation running the MSSQL database to import from. If the **Host** list is empty, you may type in the name. See your DBA or network administrator for information on how the host is specified.

#### Database

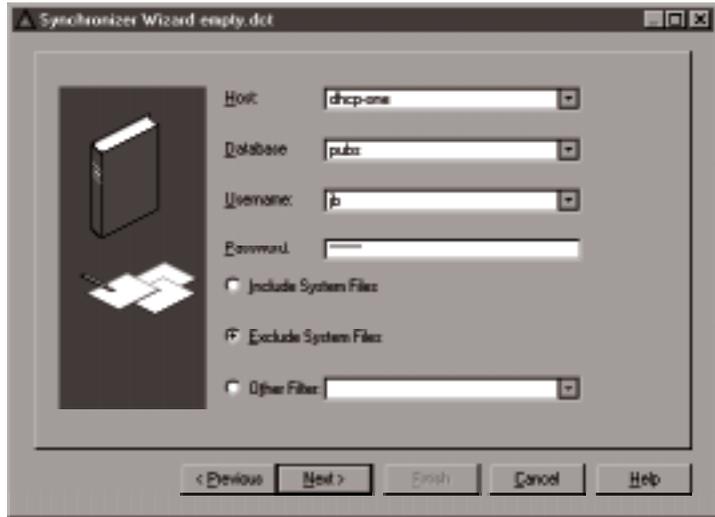
Select the MSSQL database that contains the tables or views to import. If the **Database** list is empty, you may type in the name. See your server documentation or your DBA for information on database names.

#### Username

For Standard Security, type your MSSQL Username. For Trusted Security (Integrated NT Security) no Username is required. See your server documentation or your DBA for information on applicable Usernames and security methods.

## Password

For Standard Security, type your MSSQL Password. For Trusted Security (Integrated NT Security) no Password is required. See your server documentation or your DBA for information on applicable Passwords and security methods.



### Include System Files

Select this option to include system tables in the list of importable objects.

### Exclude System Files

Select this option to exclude system tables from the list of importable objects.

### Other Filter

Select this option to provide a filter expression to limit the list of tables and views to import. The filter expression queries the `dbo.sysobjects` table. The filter expression is limited to 1024 characters in length.

**Tip:** The filter is case sensitive, so type your filter value accordingly.

Following is a list of the column names (and their Clarion datatypes) you can reference in your filter expression. Generally, filtering on MSSQL system tables requires not only an intimate knowledge of the MSSQL system tables, but also of the MSSQL stored procedures. For example, to filter on table owner:

```
uid = user_id('DBO')
```

See your SQL server documentation for information on the MSSQL system tables and stored procedures.

|            |             |
|------------|-------------|
| name       | CSTRING(31) |
| id         | LONG        |
| uid        | SHORT       |
| type       | STRING(2)   |
| userstat   | SHORT       |
| sysstat    | SHORT       |
| indexdel   | SHORT       |
| schema_ver | SHORT       |
| refdate    | STRING(8)   |
| crdate     | STRING(8)   |
| version    | LONG        |
| deltrig    | LONG        |
| instrig    | LONG        |
| updtrig    | LONG        |
| seltrig    | LONG        |
| category   | LONG        |
| cache      | SHORT       |



# 37 - ODBC ACCELERATOR

## Overview

The ODBC Accelerator Driver is one of several TopSpeed SQL Accelerator drivers. These SQL Drivers share a common code base and many common features such as TopSpeed's unique, high speed buffering technology, common driver strings, and SQL logging capability. **See *SQL Accelerator Drivers* for information on these common features.**

The ODBC Accelerator Driver converts standard Clarion file I/O statements and function calls into optimized SQL statements, which it sends to the backend SQL server for processing. This means you can use the same Clarion code to access both SQL tables and other file systems such as TopSpeed files. It also means you can use Clarion template generated code with your SQL databases.

The ODBC Accelerator Driver is slightly different from the other SQL drivers in that it is a *generic* SQL driver. It is not specific to a particular SQL server, but, in fact, works with any database or file system that supports the ODBC standard. This includes SQL systems such as AS400, Informix, MSSQL, Oracle, Scalable SQL, SQL Anywhere, Sybase, and many non-SQL systems as well (dBase, Excel, FoxPro, etc.). This chapter describes special issues and considerations that arise when using the ODBC Accelerator Driver to access data.

All the common behavior of all the SQL Accelerator drivers is documented in the *SQL Accelerators* chapter. All behavior specific to the ODBC driver is noted in this chapter.

**Note:** You must have Microsoft ODBC 2.1 or higher to access 32-bit data sources with Clarion's Database Manager or with a 16-bit application. Further, if the 32-bit data source is a Microsoft data source, you must also run Windows NT to access it with Clarion's Database Manager or with a 16-bit application.

You can download Microsoft ODBC 2.1 from [ftp.sunet.se/ftp/pub/vendor/Microsoft/developr/ODBC/public/ODBC21.exe](ftp://ftp.sunet.se/ftp/pub/vendor/Microsoft/developr/ODBC/public/ODBC21.exe).

## What is ODBC

---

ODBC (Open DataBase Connectivity) is a Windows “strategic interface” for accessing data from a variety of Relational Database Management Systems (RDBMS) across a variety of networks and platforms.

The ODBC standard was developed and is maintained by Microsoft, which publishes an ODBC Software Development Kit (SDK), geared for use with its Visual C++ product. ODBC support is another way in which Clarion provides an extensible platform for you to create applications.

## ODBC Pros and Cons

---

Using ODBC offers the following advantages:

- ◆ ODBC is an excellent choice in a Client-Server environment, especially if the Server is a native Structured Query Language (SQL) DBMS. It lets you add Client-Server support to your application, without having to do much more than choose a file driver. ODBC was specifically designed to create a non-vendor-specific method of connecting front end applications to back end services. With ODBC, the Server can handle much of the work, especially for SQL JOIN and PROJECT operations, thereby speeding up your application.
- ◆ Existing ODBC drivers cover a great many types of databases. There are ODBC drivers available for databases for which Clarion may not have a native driver—for example, for Microsoft Excel and Lotus Notes files.
- ◆ ODBC is already widespread. Major application suites such as Microsoft Office install ODBC drivers for file formats such as dBase and Microsoft Access. Keep in mind that many ODBC back end drivers have been updated and you should obtain the latest releases.
- ◆ ODBC is platform independent. One of Microsoft’s prime objectives in establishing ODBC was to support easier access to legacy systems, or corporate environments where data resides on diverse platforms or multiple DBMS’s. As long as an ODBC driver and back end are available, it doesn’t matter whether you use Microsoft’s NetBEUI, SPX/IPX, DECNet or others; your application can connect to the DBMS and access the data.

Given that there are many drivers available, and that the standard was developed by the company that developed Windows, you might consider using ODBC as the driver of choice for *all* your Windows applications. Yet, when deciding whether to use an ODBC driver or a Clarion native database driver, you must also consider possible disadvantages:

- ◆ ODBC adds a layer—the ODBC Driver Manager—between your

application and the database. When accessing files on a local hard drive, this generally results in slower performance. The driver manager must translate the application's ODBC API call to an SQL statement before any data access occurs.

ODBC uses SQL to communicate with the back end database. Although this can be very efficient when communicating with Client/Server database engines, it is normally less efficient than direct record access when using a file system designed around single record access, such as xBase or Btrieve.

- ◆ The ODBC Administrator manipulates the Windows registry in 32bit and the ODBC.INI in 16bit, adding complexity to systems that run under both 16-bit and 32-bit operation systems.
- ◆ The information required by the ODBC database manager to connect to a data source varies from one ODBC driver to another. Unlike the selection of Clarion file drivers, where file operations are virtually transparent, you may need to do some work to gather the information required to use a particular ODBC driver. This chapter provides a few tips that might make it easier, and many ODBC drivers come with a Help (.HLP) file which documents special settings usually stored in ODBC.INI (16-bit only); but the burden is on *you* to solve any problems with third-party ODBC drivers.
- ◆ ODBC is not included with Windows. When distributing your application, you'll need to install the ODBC drivers and the ODBC driver manager into the end user's system. This requires the ODBC SDK from Microsoft. In some cases, the back end server may have already provided a distribution kit which installs the ODBC driver on the workstation.
- ◆ The normal Microsoft setup program that installs the ODBC driver manager adds an applet to the end user's Control Panel window for managing ODBC. It's very easy for an end user to use this tool to change the settings in the ODBC.INI file (16-bit only). The end user can unwittingly remove or modify the settings for the back end ODBC driver which would make it impossible for your application to connect to the data file.

Given the pros and cons, we recommend using the native Clarion file drivers when both a native driver and an ODBC driver exist for the same file format.

## How ODBC Works

---

When you use ODBC to access data, four components must cooperate to make it work:

- ◆ *Your application* calls the ODBC driver manager, and sends it the appropriate requests for data, with the ODBC API.

Clarion does this for you transparently, using either the C4ODB.DLL (16-bit) or the C4ODBX.DLL (32-bit) application extension. When hand-coding, be sure to include this library in the project. When distributing your application, be sure to deploy this file with your .EXE file (unless you produce a one-piece .EXE).

- ◆ *The ODBC driver manager* receives the API calls, checks ODBC.INI (16-bit) or the Windows Registry (32-bit) for information on the data source, then loads the ODBC “back-end” driver.

The actual “interface” to the driver manager is a file called ODBC.DLL (16-bit) and ODBC32.DLL (32-bit), which the Microsoft setup program places in the \Windows\System directory. This is the ODBC Administrator, which then loads other libraries to do its work.

- ◆ *The ODBC “backend” driver* is another library (.DLL) which contains the executable code for accessing the data.

Various third-parties supply “backend” drivers. For example, Lotus Development Corp. supplies the ODBC driver for Lotus Notes. Microsoft Office distributes an ODBC SDK containing drivers for most of their database products.

- ◆ *The data source* is either a data file (usually when ODBC is used for local data access), or a remote DBMS, such as an Oracle database.

The data source has a descriptive name; for example, “Microsoft Access Databases.” The name serves as the section name in the ODBC.INI file.

The ODBC driver manager *must* know the exact data source name so that it can load the right driver to access the data. Therefore, it's vitally important that *you* know the precise data source name.

## ODBC Data Types

| Clarion Data Types |                     |         |      |       |      |       |       |      |         |          |      |      |
|--------------------|---------------------|---------|------|-------|------|-------|-------|------|---------|----------|------|------|
| ODBC Data Types    | STRING              | CSTRING | BYTE | SHORT | LONG | ULONG | SREAL | REAL | DECIMAL | PDECIMAL | DATE | TIME |
| M                  | CHAR                | C       | •    | •     | •    | •     | •     | •    | •       | •        |      |      |
| I                  | VARCHAR             | •       | C    | •     | •    | •     | •     | •    | •       | •        |      |      |
| N                  | LONG<br>VARCHAR     | •       | •    |       |      |       |       |      |         |          |      |      |
|                    | DECIMAL             |         | •    | •1    | •1   | •1    |       | •4   | •4      |          | •    |      |
|                    | NUMERIC             |         | •    | •1    | •1   | •1    |       | •4   | •4      | •        |      |      |
| C                  | SMALLINT            |         | •    | •3    | C5   | •3    | •3    |      |         |          |      |      |
| O                  | INTEGER             |         | •    | •3    | •3   | C5    | •3    |      |         |          |      |      |
| R                  | REAL                |         | •    |       |      |       |       | C    | •3      | •4       | •4   |      |
| E                  | FLOAT               |         | •    |       |      |       |       | •3   | •       | •4       | •4   |      |
|                    | DOUBLE<br>PRECISION |         | •    |       |      |       |       | •3   | C       | •4       | •4   |      |
| E                  | BIT                 |         | •    | •     | •3   | •3    | •3    |      |         |          |      |      |
| X                  | TINYINT             |         | •    | C5    | •3   | •3    | •3    |      |         |          |      |      |
| T                  | BIGINT              |         | •    |       | •3   | •3    | •3    | •3   | •3      | •        | •    |      |
| E                  | BINARY              | •       |      |       |      |       |       |      |         |          |      |      |
| N                  | VARBINARY           | •       |      |       |      |       |       |      |         |          |      |      |
| D                  | LONG<br>VARBINARY   | •       |      |       |      |       |       |      |         |          |      |      |
| E                  | DATE                |         |      |       |      |       |       |      |         |          | C    |      |
| D                  | TIME                |         |      |       |      |       |       |      |         |          |      | C    |
|                    | TIMESTAMP           | •2      |      |       |      |       |       |      |         |          |      |      |

### Notes

- C The Clarion data type can be used to manipulate the ODBC data type. CREATE does create the ODBC data type.
- The Clarion type can be used to manipulate the ODBC data type, however, CREATE does NOT create the ODBC data type.
- 1 Clarion LONG, SHORT, and BYTE can be used with ODBC DECIMAL and NUMERIC data types if the ODBC field does not have any decimal places.
  - 2 ODBC TIMESTAMP fields can be manipulated using a STRING(8) followed by a GROUP over it which contains only a DATE field and a TIME field.

**Example:**

```
TimeStampField STRING(8),NAME('TimeStampField')
TimeStampGroup GROUP,OVER(TimeStampField)
TimeStampDate DATE
TimeStampTime TIME
END
```

CREATE creates a TIMESTAMP field if you use a similar structure.

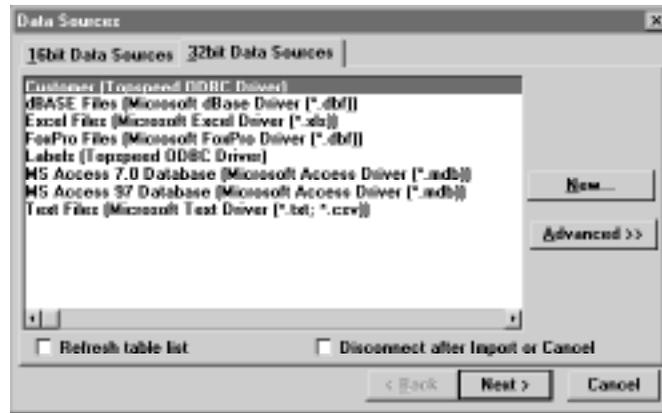
- 3 Some loss of precision may occur.
- 4 Rounding errors may occur.
- 5 CREATE attempts to create a TINYINT for a BYTE. If the backend does not support TINYINT, CREATE treats BYTE as a SHORT. CREATE attempts to create a SMALLINT for a SHORT. If the backend does not support SMALLINT, CREATE treats SHORT as a LONG. CREATE attempts to create an INTEGER for a LONG. If the backend does not support INTEGER, CREATE creates a decimal field.

**Note:** Your backend database may contain data types that are not listed here. These data types are converted to ODBC data types by the backend database. Consult your backend database documentation to determine which ODBC data type is used.

## Importing from ODBC Data Sources

Clarion's Dictionary Editor Import Wizard lets you import table definitions into your Clarion Data Dictionary.

When you select the ODBC Accelerator Driver from the driver drop-down list, the Import Wizard opens the **Data Sources** dialog. Select an existing Data Source, then press the **Next** button to import its definition.



If the data source is not defined, you can add it by pressing the **New** button, then following the ODBC instructions provided by the file system you wish to access.

When you have selected a data source, press the **Next** button to import its definition. This imports the ODBC table definition and opens the **File Properties** dialog, allowing you to modify file attributes, if you choose.

### Connection Information and Driver Configuration—File Properties

---

Typically, you add SQL support to your application by importing the SQL or ODBC table, view, and synonym definitions into your Clarion Data Dictionary. The Import Wizard automatically fills in the **File Properties** dialog with default values based on the imported item. However, there are several fields in the **File Properties** dialog you can use to further configure the way the ODBC Accelerator Driver accesses the data. These **File Properties** fields, and their uses are described below.

#### Owner Name

Typically, the Import Wizard places the SQL database connection information (Host, Username, Password, etc.) in the **Owner Name** field.

Some backend databases may require additional connection information which you can supply in the **Owner Name** field. This information follows the password and is separated by semicolons, using the syntax:  
*keyword=value;keyword=value.*

For example, when accessing a Sybase database with the ODBC driver, this would appear as:

```
DataSource,UserID,PassWord,DATABASE=DataBaseName;APP=APPName
```

Consult your SQL Server's documentation for information on these keywords, their uses and effects.

## Key Configuration—Key Properties

---

Typically, you add SQL support to your application by importing the SQL or ODBC table, view, and synonym definitions into your Clarion Data Dictionary. The Import Wizard automatically fills in the **Key Properties** dialog with default values based on the imported item. However, there are some fields in the **Key Properties** dialog you can use to further configure the way the ODBC Accelerator Driver accesses the data. These **Key Properties** fields are described below. where clauses that are invalid for the backend server.

### External Name

#### | READONLY

Adding the READONLY switch to the **External Name** tells the ODBC driver not to insert the field when the record is added. This is necessary for certain backends (such as Watcom) that do not allow autoincrementing key fields to be set to null. Some backends do not allow autoincrementing key fields to be set to null, but they fail to advise the ODBC Accelerator Driver of this restriction. The READONLY switch lets you manually advise of the restriction.

## Column Configuration—Field Properties

---

Typically, you add SQL support to your application by importing the SQL or ODBC table, view, and synonym definitions into your Clarion Data Dictionary. The Import Wizard automatically fills in the **Field Properties** dialog with default values based on the imported item. However, there are some fields in the **Field Properties** dialog you can use to further configure the way the SQL Accelerator Driver accesses the data. These **Field Properties** fields are described below.

## External Name

### | NOWHERE

Adding the NOWHERE switch to the **External Name** tells the ODBC driver to exclude the field from any where clauses it sends to the backend server. This is necessary for certain backends when WATCH is in effect. Some backends do not allow certain data types in a where clause, but they fail to advise the ODBC Accelerator Driver of this restriction. The NOWHERE switch lets you manually advise of the restriction when WATCH causes the ODBC driver to generate

## ***Debugging Your ODBC Application***

When you use the ODBC Accelerator Driver, the ODBC Administrator can create a log file documenting all calls made by the ODBC Accelerator Driver. It includes the actual SQL statements made by the ODBC driver to the data source, and includes any errors posted. This administrator logging slows down your program considerably, so it should only be activated during testing. Additionally, the log file can grow to large proportions very quickly, so you should turn logging off and delete the log file after using it.

Besides “snooping” on the actual SQL statements generated by the driver, you can zero in on any errors. For example, if the application is unable to connect, you can open the log file, scroll to the bottom of the file, then work up until you find the word “SQLError.”

See Microsoft’s ODBC documentation (ODBCINST.HLP—*ODBC Options Dialog Box*) for instructions on using the ODBC Administrator logs.

## Driver Strings

There are switches or “driver strings” you can set to control the way your application creates, reads, and writes files with a specific driver. Driver strings are simply messages or parameters that are sent to the file driver at run-time to control its behavior. See *Common Driver Features—Driver Strings* for an overview of these runtime Database Driver switches and parameters.

**Tip:** A forward slash precedes all SQL Accelerator driver strings. The slash allows the driver to distinguish between driver strings and SQL statements sent with SEND.

In addition to the standard SQL Driver Strings, the ODBC Accelerator Driver supports the following Driver Strings:

### BINDCONSTANTS

```
DRIVER('ODBC', '/BINDCONSTANTS = TRUE | FALSE ')
[Bind" =] SEND(file, '/BINDCONSTANTS [= TRUE | FALSE]')
```

By default (BINDCONSTANTS=TRUE) the ODBC Accelerator binds memory locations instead of generating text equivalents for constant values. However, some backends get confused when doing this. So if you find that your ODBC based BrowseBox will not scroll, or your backend returns incorrect results for a BrowseBox you can turn off binding of constant values by setting BINDCONSTANTS to FALSE.

### CLIPSTRINGS

```
DRIVER('ODBC', '/CLIPSTRINGS = TRUE | FALSE ')
[Clipped" =] SEND(file, '/CLIPSTRINGS [= TRUE | FALSE]')
```

By default (CLIPSTRINGS=TRUE), the ODBC driver CLIPs strings before sending them to the backend server (see *CLIP* in the *Language Reference*). To send the full (unclipped) string, set CLIPSTRINGS=FALSE.

### FORCEUPPERCASE

```
DRIVER('ODBC', '/FORCEUPPERCASE = TRUE | FALSE ')
[Uppered" =] SEND(file, '/FORCEUPPERCASE [= TRUE | FALSE]')
```

By default (FORCEUPPERCASE=FALSE), the ODBC Driver passes the table name in mixed case to the SQLColumns function to verify the existence of the table. However, some backends require the table name to be passed in uppercase. To pass the table name in uppercase, set CLIPSTRINGS=TRUE. See also *VERIFYVIASELECT*.

## GATHERATOPEN

---

```
DRIVER('AS400', '/GATHERATOPEN = TRUE | FALSE ')
```

By default the driver delays gathering field information until it is required. However, some backends (like Sybase 11) perform poorly under these conditions. Setting `GATHERATOPEN` to `TRUE` forces the driver to gather most of the field information when the file is opened, which avoids a slowdown during program execution.

## JOINTYPE

---

```
DRIVER('ODBC', '/JOINTYPE = Watcom | DB2 | Microsoft | FirstSQL | Inner | None')
[Join" =] SEND(file, '/JOINTYPE [= Watcom | DB2 | Microsoft | FirstSQL | Inner | None]')
```

The ODBC 2.1 standard does not support joins to more than one child (or parent). Most vendors consider this limitation unacceptable and have extended the standard. However, they have done so in different ways. The ODBC driver attempts to determine the join type used by the backend, but if it does not get it right, then you should use the `JOINTYPE` driver string in the primary file of the view. Note that specifying *Inner* is normally slower than *Watcom*, *DB2*, *Microsoft* or *FirstSQL* and *None* is slower than *Inner*, but will work with all backends because the join is done on the client.

The ODBC 3.0 standard does support multiple joins, so ODBC 3.0 compliant drivers should not require this switch.

## NESTING

---

```
DRIVER('ODBC', '/NESTING = TRUE | FALSE ')
[Nest" =] SEND(file, '/NESTING [= TRUE | FALSE]')
```

Some ODBC drivers do not support parent->child->grandchild style views. The ODBC driver attempts to determine if this is supported. If the driver does not get it right and the backend does not support these type of views, then you need to set `NESTING=FALSE`. This causes the join to be done on the client.

## ODBCCALL

---

```
DRIVER('ODBC', '/ODBCCALL = TRUE | FALSE ')
[Call" =] SEND(file, '/ODBCCALL [= TRUE | FALSE]')
```

By default (`ODBCCALL = True`) the ODBC Accelerator reformats your `CALL` statements to match the ODBC standard call syntax. To disable this automatic reformatting, set `ODBCCALL=FALSE`.

## USEINNERJOIN

---

**DRIVER('ODBC', '/USEINNERJOIN= TRUE | FALSE' )  
[ Join" = ] SEND(file, '/USEINNERJOIN [ = TRUE | FALSE ]')**

By default (USEINNERJOIN = True) the ODBC Accelerator generates the following ANSI SQL for inner joins:

```
SELECT ... FROM table1 INNER JOIN table2 ON table1.field=table2.field
```

However, not all backends support ANSI SQL. The driver provides an alternative syntax for inner joins. To generate the following alternative syntax, set USEINNERJOIN=FALSE:

```
SELECT ... FROM table1, table2 WHERE table1.field=table2.field
```

## VERIFYVIASELECT

---

**DRIVER('ODBC', '/VERIFYVIASELECT = TRUE | FALSE' )  
[ Verify" = ] SEND(file, '/VERIFYVIASELECT [ = TRUE | FALSE ]')**

VERIFYVIASELECT lets the ODBC Driver use an alternative, sometimes faster, method to validate fields when opening a table. By default (VERIFYVIASELECT=FALSE), the ODBC Driver uses the SQLColumns function to validate fields. However, some backends (particularly SQL Anywhere) can validate fields faster using a SELECT statement. To verify fields using the SELECT statement, set VERIFYVIASELECT to TRUE.

VERIFYVIASELECT defaults to TRUE for SQL Anywhere backends.

## ZEROISNULL

---

**DRIVER('ODBC', '/ZEROISNULL = TRUE | FALSE' )  
[ Nulls" = ] SEND(file, '/ZEROISNULL [ = TRUE | FALSE ]')**

ZEROISNULL lets the ODBC Accelerator Driver set DATE and TIME fields to zero (0) rather than null. By default (ZEROISNULL=TRUE), the ODBC Accelerator Driver assumes a DATE or TIME field with a value of zero (0) should be a null value in the backend database, and adjusts the values to NULL when writing to the backend. To allow the driver to set DATE and TIME fields to zero rather than null, set ZEROISNULL to FALSE.

## Driver Properties

You can use Clarion's property syntax to query and set certain ODBC Accelerator properties. In addition to the standard SQL Accelerator properties (see *SQL Accelerator Drivers—SQL Accelerator Properties*), the ODBC Accelerator supports the following properties.

---

### PROP:AlwaysRebind

PROP:AlwaysRebind sets or returns the toggle that controls whether the ODBC Accelerator rebinds memory locations when a NULL state changes.

For all backends except MSSQL, PROP:AlwaysRebind defaults to 0 or False, so the ODBC driver does not rebind memory locations when a NULL state changes. However, some SQL backends (including MSSQL) do not recheck the null state, so must have the memory location rebound to force the change of null state. Setting PROP:AlwaysRebind to 1 or True tells the ODBC Accelerator to do this extra binding.

---

### PROP:hdbc

PROP:hdbc returns the current hdbc used by the ODBC driver. Thus ?MyFile{PROP:hdbc} may be used for ODBC API calls requiring hdbc.

---

### PROP:henv

PROP:henv returns the current henv used by the ODBC driver. Thus ?MyFile{PROP:henv} may be used for ODBC API calls requiring henv. For example, the SQLDescribeCol function:

```
rc# = SQLDataSources(Myfile{PROP:henv},SQL_FETCH_NEXT,ODBC:driver, |
 drvLen,drvLen,ODBC:Description,desclen,desclen)
```

---

### PROP:hstmt

PROP:hstmt returns the current hstmt used by the ODBC driver. Thus ?MyFile{PROP:hstmt} may be used for ODBC API calls requiring hstmt. For example, the SQLDescribeCol function:

```
Myfile{PROP:SQL} = 'Select * from ATable'
rc# = SQLDescribeCol(Myfile{PROP:hstmt},Num,Name,Max,NameL, |
 Type,Def,Scale,Null)
```

## PROP:LoginTimeout

---

PROP:LoginTimeout sets a time limit in seconds for an SQL database's login screen. If the user does not respond in the allotted time, the connection fails and the login is aborted. The default is to wait indefinitely for user input. Some servers do not support this feature and may ignore the instruction. For example:

```
AFile FILE,DRIVER('ODBC'),OWNER('DataSource')
CODE
OPEN(Afile)
IF NOT ERRORCODE()
 Afile{PROP:LoginTimeOut}=60 !allow 1 minute for login
END
```

## PROP:LogonScreen

---

PROP:LogonScreen sets or returns the toggle that determines whether the driver automatically prompts for logon information. By default (PROP:LogonScreen=True), the driver displays a logon window if needed (if no connect string—host, username, and password—is supplied). To suppress the automatic logon window, set PROP:LogonScreen to zero (0 or False). If no connect string is supplied, the OPEN(file) fails and FILEERRORCODE() returns '08001' or '28000.' For example:

```
AFile FILE,DRIVER('ODBC'),OWNER('DataSource')
CODE
Afile{PROP:LogonScreen}=0
OPEN(Afile)
```

## PROP:QuoteString

---

PROP:QuoteString sets or returns the column name delimiter (typically a quote) the ODBC Accelerator Driver uses to surround column names within its generated SQL statements. Different backends require different delimiter characters.

You can use PROP:QuoteString to build your own dynamic SQL statements. Note that you must enclose any column names that are also SQL reserved words in the correct delimiter character. See *Using Embedded SQL*.

Some backends do not correctly return the delimiter character. For those backends you should set the value of PROP:QuoteString before using it.

## ***Using Embedded SQL***

You can use Clarion's property syntax (PROP:SQL) to send SQL statements to the backend SQL server within the normal execution of your program. See *SQL Accelerators—Using Embedded SQL* for more information.

## **Calling a Stored Procedure**

---

For ODBC, NORESULTCALL is more efficient than CALL and may be required for some backends.

By default, the ODBC Accelerator reformats CALL statements to the standard ODBC syntax for CALL statements. See *Driver Strings—ODBCCALL* for more information.

## Supported Attributes and Procedures

### File Attributes Supported

|                                                          |                |
|----------------------------------------------------------|----------------|
| CREATE .....                                             | Y              |
| DRIVER( <i>filetype</i> [, <i>driver string</i> ]) ..... | Y              |
| NAME .....                                               | Y              |
| ENCRYPT .....                                            | N              |
| OWNER( <i>password</i> ) .....                           | Y <sup>2</sup> |
| RECLAIM .....                                            | N              |
| PRE( <i>prefix</i> ) .....                               | Y              |
| BINDABLE .....                                           | Y              |
| THREAD .....                                             | Y <sup>6</sup> |
| EXTERNAL( <i>member</i> ) .....                          | Y              |
| DLL( <i>flag</i> ) .....                                 | Y              |
| OEM .....                                                | N <sup>3</sup> |

### File Structures Supported

|              |                |
|--------------|----------------|
| INDEX .....  | Y <sup>3</sup> |
| KEY .....    | Y <sup>3</sup> |
| MEMO .....   | N              |
| BLOB .....   | N              |
| RECORD ..... | Y              |

### Index, Key, Memo Attributes Supported

|                             |                |
|-----------------------------|----------------|
| BINARY .....                | N <sup>7</sup> |
| DUP .....                   | Y              |
| NOCASE .....                | Y              |
| OPT .....                   | N              |
| PRIMARY .....               | Y              |
| NAME .....                  | Y              |
| Ascending Components .....  | Y              |
| Descending Components ..... | Y              |
| Mixed Components .....      | Y              |

### Field Attributes Supported

|            |   |
|------------|---|
| DIM .....  | N |
| OVER ..... | Y |
| NAME ..... | Y |

### File Procedures Supported

|                                                 |                |
|-------------------------------------------------|----------------|
| BOF( <i>file</i> ) .....                        | N              |
| BUFFER( <i>file</i> ) .....                     | Y              |
| BUILD( <i>file</i> ) .....                      | Y              |
| BUILD( <i>key</i> ) .....                       | Y              |
| BUILD( <i>index</i> ) .....                     | Y <sup>8</sup> |
| BUILD( <i>index, components</i> ) .....         | Y <sup>8</sup> |
| BUILD( <i>index, components, filter</i> ) ..... | N              |
| BYTES( <i>file</i> ) .....                      | Y              |
| CLOSE( <i>file</i> ) .....                      | Y              |
| COPY( <i>file, new file</i> ) .....             | N              |
| CREATE( <i>file</i> ) .....                     | Y              |
| DUPLICATE( <i>file</i> ) .....                  | Y              |
| DUPLICATE( <i>key</i> ) .....                   | Y              |
| EMPTY( <i>file</i> ) .....                      | Y              |
| EOF( <i>file</i> ) .....                        | N              |
| FLUSH( <i>file</i> ) .....                      | N              |
| LOCK( <i>file</i> ) .....                       | N              |
| NAME( <i>label</i> ) .....                      | Y              |
| OPEN( <i>file, access mode</i> ) .....          | Y              |

|                                         |   |
|-----------------------------------------|---|
| PACK( <i>file</i> ) .....               | N |
| POINTER( <i>file</i> ) .....            | N |
| POINTER( <i>key</i> ) .....             | N |
| POSITION( <i>file</i> ) .....           | N |
| POSITION( <i>key</i> ) .....            | Y |
| RECORDS( <i>file</i> ) .....            | Y |
| RECORDS( <i>key</i> ) .....             | Y |
| REMOVE( <i>file</i> ) .....             | Y |
| RENAME( <i>file, new file</i> ) .....   | N |
| SEND( <i>file, message</i> ) .....      | Y |
| SHARE( <i>file, access mode</i> ) ..... | Y |
| STATUS( <i>file</i> ) .....             | Y |
| STREAM( <i>file</i> ) .....             | N |
| UNLOCK( <i>file</i> ) .....             | N |

### Record Access Supported

|                                               |                |
|-----------------------------------------------|----------------|
| ADD( <i>file</i> ) .....                      | Y              |
| ADD( <i>file, length</i> ) .....              | N              |
| APPEND( <i>file</i> ) .....                   | Y              |
| APPEND( <i>file, length</i> ) .....           | N              |
| DELETE( <i>file</i> ) .....                   | Y              |
| GET( <i>file, key</i> ) .....                 | Y              |
| GET( <i>file, filepointer</i> ) .....         | N              |
| GET( <i>file, filepointer, length</i> ) ..... | N              |
| GET( <i>key, keypointer</i> ) .....           | N              |
| HOLD( <i>file</i> ) .....                     | N              |
| NEXT( <i>file</i> ) .....                     | Y              |
| NOMEMO( <i>file</i> ) .....                   | N              |
| PREVIOUS( <i>file</i> ) .....                 | Y <sup>4</sup> |
| PUT( <i>file</i> ) .....                      | Y              |
| PUT( <i>file, filepointer</i> ) .....         | N              |
| PUT( <i>file, filepointer, length</i> ) ..... | N              |
| RELEASE( <i>file</i> ) .....                  | N              |
| REGET( <i>file, string</i> ) .....            | N              |
| REGET( <i>key, string</i> ) .....             | Y              |
| RESET( <i>file, string</i> ) .....            | N              |
| RESET( <i>key, string</i> ) .....             | Y              |
| SET( <i>file</i> ) .....                      | Y <sup>4</sup> |
| SET( <i>file, key</i> ) .....                 | N              |
| SET( <i>file, filepointer</i> ) .....         | N              |
| SET( <i>key</i> ) .....                       | Y              |
| SET( <i>key, key</i> ) .....                  | Y              |
| SET( <i>key, keypointer</i> ) .....           | N              |
| SET( <i>key, key, filepointer</i> ) .....     | N              |
| SKIP( <i>file, count</i> ) .....              | Y              |
| WATCH( <i>file</i> ) .....                    | Y              |

### Transaction Processing Supported<sup>5</sup>

|                                                 |                |
|-------------------------------------------------|----------------|
| LOGOUT( <i>timeout, file, ..., file</i> ) ..... | Y <sup>9</sup> |
| COMMIT .....                                    | Y              |
| ROLLBACK .....                                  | Y              |

### Null Data Processing Supported

|                                  |   |
|----------------------------------|---|
| NULL( <i>field</i> ) .....       | Y |
| SETNULL( <i>field</i> ) .....    | Y |
| SETNONNULL( <i>field</i> ) ..... | Y |

## Notes

---

- 1 The Clarion ODBC file driver supports the listed items, however, the underlying file system may not support all of these items.
- 2 We recommend using a variable password that is lengthy and contains special characters because this more effectively hides the password value from anyone looking for it. For example, a password like “dd...#\$...\*&” is much more difficult to “find” than a password like “SALARY.”

**Tip:** To specify a variable instead of the actual password in the Owner Name field of the File Properties dialog, type an exclamation point (!) followed by the variable name. For example: !MyPassword.

- 3 International sorting is assumed to be done by the underlying file system. As such the OEM attribute and the .ENV file are ignored.
- 4 PREVIOUS is not supported in file order.
- 5 See also *PROP:Logout* in the *Language Reference*.
- 6 THREADED files do not consume additional file handles for each thread that accesses the file.
- 8 BUILD(index) sets internal driver flags to guarantee the driver generates the correct ORDER BY clause. The driver does not call the backend server.
- 9 Whether LOGOUT also LOCKs the table depends on the server’s configuration for transaction processing. See your server documentation..

## **Microsoft Access and ODBC**

### **ODBC Driver that ships with Access 2.0**

The ODBC driver that ships with Access 2.0 only works with other Microsoft Office applets. To get a general purpose driver that works with the Clarion ODCB Accelerator Driver, you need to purchase the ODBC Desktop Driver Kit 2.0 from Microsoft.

### **Accessing 32-Bit ODBC Data Sources from 16-Bit Applications**

To access 32-bit ODBC drivers such as the Access 7.0 or Access 97 drivers from a 16-bit application, you must use ODBC.DLL version 2.1 or later.

**Note:** You must have Microsoft ODBC 2.1 or higher to access 32-bit data sources with Clarion's Database Manager or with a 16-bit application. Further, if the 32-bit data source is a Microsoft data source, you must also run Windows NT to access it with Clarion's Database Manager or with a 16-bit application.

You can download Microsoft ODBC 2.1 from <ftp://ftp.sunet.se/pub/vendor/microsoft/developr/ODBC/public/ODBC21.exe>

## Configurable **ERROR** Messages

All of TopSpeed's database drivers post error conditions and messages that can be accessed with the `ERRORCODE()`, `ERROR()`, `FILEERRORCODE()` and `FILEERROR()` procedures (see the *Language Reference*). The drivers post the codes immediately after each I/O (`OPEN`, `NEXT`, `GET`, `ADD`, `DELETE`, etc.) operation.

**Tip:** The `ERRORFILE()` procedure returns the file or table that produced the error.

See *Run Time Errors* in the *Language Reference* for information on the `ERRORCODE()` values and their corresponding `ERROR()` message text. The `ERROR()` text is configurable using the environment file (`CLAMSG`) and the `LOCALE` procedure. See *Internationalization*, `CLAMSG`, and `LOCALE` in the *Language Reference* for more information.

In addition, ODBC Accelerator `FILEERROR()` message text is also configurable using the environment file (`CLAMSG`) and the `LOCALE` procedure. However, the `CLAMSG` value required to change the message text is equal to the `FILEERRORCODE()` value + 4000. Following is a list of the `FILEERROR()` text, the `FILEERRORCODE()` values, and the `CLAMSG` values required to change the `FILEERROR()` text:

| <b>FILEERRORCODE</b> | <b>CLAMSG</b> | <b>FILEERROR</b>             |
|----------------------|---------------|------------------------------|
| 100                  | 4100          | ODBC.DLL Could Not Be Loaded |



# 38 - ORACLE ACCELERATOR

## Overview

### Oracle Server

---

For complete information on the Oracle database system, please refer to your Oracle documentation.

### Oracle Accelerator

---

Oracle Accelerator is one of several TopSpeed SQL Accelerators. These SQL drivers share a common code base and many common features such as TopSpeed's unique, high speed buffering technology, common driver strings, and SQL logging capability. **See *SQL Accelerators* for information on these common features.**

The Oracle Accelerator converts standard Clarion file I/O statements and function calls into optimized SQL statements, which it sends to the backend Oracle server for processing. This means you can use the same Clarion code to access both Oracle tables and other file systems such as TopSpeed files. It also means you can use Clarion template generated code with your SQL databases.

All the common behavior of all the SQL Accelerator drivers is documented in the *SQL Acceleratos* chapter. All behavior specific to the Oracle Accelerator is noted in this chapter.

TopSpeed's Oracle Accelerator automatically works with Oracle versions 7.0 through 8.03. At runtime, the driver initially tries to load the Oracle 8.03 DLLs. If the 8.03 DLLs are not available, it tries the Oracle 8.0 DLLs, 7.3 DLLs, and so on. See *Future Oracle Releases* for more information.

**Note:** Personal Oracle 8.0 only works with 32-bit programs.

## System Requirements

---

### Hardware

You can run the Clarion development environment on any system that meets the minimum system requirements for Microsoft Windows 3.x, Windows 95™, or Windows NT 3.51.

## Software



To *develop* 16-bit or 32-bit Windows programs with Oracle Accelerator, you must have Oracle version 7.0 or higher; that is, you must have a licensed copy of Oracle's **ORAxWIN.DLL** (where x is the Oracle version number), plus any other DLLs it requires—typically Oracle's USDMEM.DLL and COREWIN.DLL. **These .DLLs must be in a directory that is in your system PATH.**

In addition, to *run* your 32-bit programs with Oracle, you must have Oracle version 7.2 or higher; that is, you must have a licensed copy of Oracle's ORA72.DLL or higher. This DLL must also be in a directory that is in your system PATH.

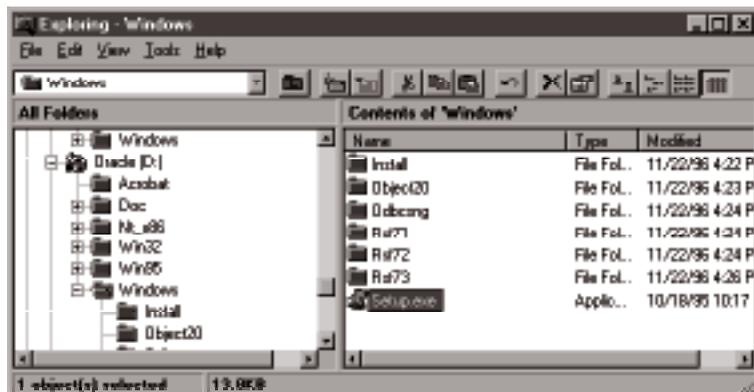
**Note:** Oracle 7.2 and higher (32-bit) does not automatically install all the .DLLs required by Clarion; you must specify Required Support Files for Windows (16-bit) in addition to the standard installation files. To do so, run SETUP.EXE from the WINDOWS directory on the Oracle CD.

**Note:** You will not be able to define or import Oracle files in your Clarion data dictionary until the Oracle DLLs are installed in a directory that is in your system PATH.

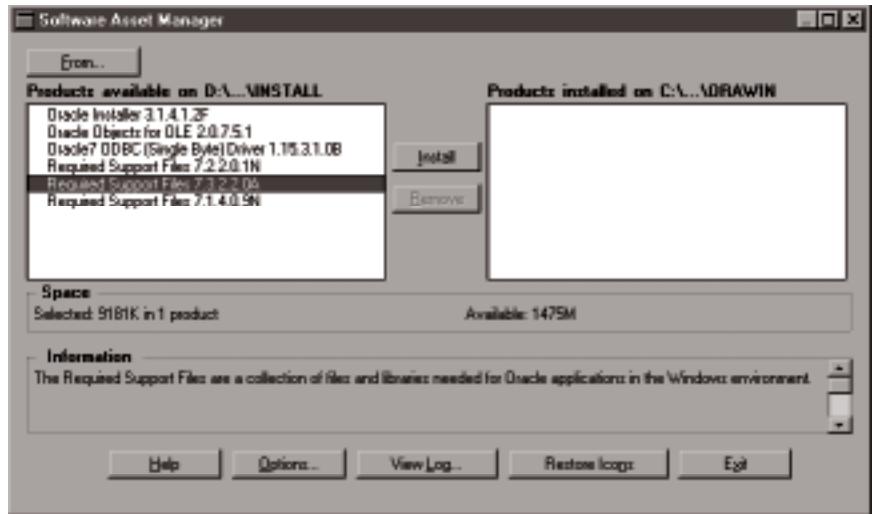
## Installing Oracle's Required Support Files

Oracle 7.2 and higher does not automatically install all the .DLLs required for Clarion with its standard install; you must install the Windows (16-bit) Required Support Files for your version of Oracle in addition to the standard installation files. You should follow Oracle's installation instructions for installing Required Support Files. We have found the following steps work best for Oracle 7.3:

1. Run SETUP.EXE from the WINDOWS directory on the Oracle CD.



2. Choose an install directory other than the primary Oracle directory.
3. Install the support files corresponding to your Oracle version only.



## Installation

- ❑ *To install the Oracle Accelerator file driver:*

1. Install the required Oracle components.  
See *System Requirements—Software*.
2. Run **A:SETUP** where A: is the drive letter of the drive containing the Oracle Accelerator install disk.

Follow the instructions on your screen. Install Oracle Accelerator to the directory that already contains Clarion.

The setup program installs Oracle Accelerator according to your selections. When everything is installed, the setup program offers to open the Oracle Accelerator on-line help file. This file contains late breaking information about the Oracle Accelerator file driver.

3. Please read the late breaking information.  
When you have finished reading, close the file.
4. Register the Oracle Accelerator driver with the Clarion development environment.

## Registering the Oracle Accelerator

---

You must register the Oracle Accelerator driver with the Clarion development environment before you can use it.

❑ *To register Oracle Accelerator:*

1. Start the Clarion development environment.
2. Choose **Setup ► Database Driver Registry**.
3. Press the **Add** button.
4. Highlight C5ORA.DLL (by default, in the ..\CLARION5\BIN directory) in the list box, then press the **OK** button.

This registers the Oracle Accelerator driver (both 16-bit and 32-bit versions).

5. Press the **OK** button.

## If You're New to Clarion...

---

Developing applications with Clarion is generally the same process, regardless of the file system you choose. Therefore, we strongly recommend that you work through the tutorials in the *Getting Started* and *Learning Clarion* books to increase your understanding of the Clarion environment generally. Then work through the tutorial in this book to focus on the special considerations that apply when you develop applications with Oracle.

Clarion has the capability (through its templates) to automatically generate lots of code for browsing, updating, and reporting your Oracle database. The template-generated code is designed to work with a variety of file systems, including Oracle; however there are a few specific exceptions you should be aware of. These specific exceptions and their work-arounds are illustrated in the tutorial in this book.

In addition to working through the tutorials, please take a few moments to review the following topics:

- ◆ *Introduction*
  - System Requirements—Software*
  - Registering the Oracle Accelerator*
- ◆ *Using Oracle Accelerator*
  - Overview*
- ◆ *SQL Accelerator Drivers*

## If You're New to Oracle...

---

Oracle is a database environment. It has many powerful tools to help you design, build, and maintain a secure, internally consistent database. We recommend that you educate yourself to use Oracle's powerful Database Administration (DBA) capabilities and features to accomplish primary database definition, creation, maintenance, integrity, and security.

You may issue Oracle database management commands through the Oracle Accelerator driver to perform DBA tasks; however we generally do not recommend using Clarion data management statements like CREATE() to manipulate table definitions. The native Oracle statements provide more control over DBA tasks (such as allocating database space, granting access privileges, etc.).

Developing applications with Clarion is generally the same process, regardless of the file system you choose, but there are some exceptions. We strongly recommend that you work through the tutorial in this book to focus on the special considerations that apply when you develop Clarion applications with Oracle.

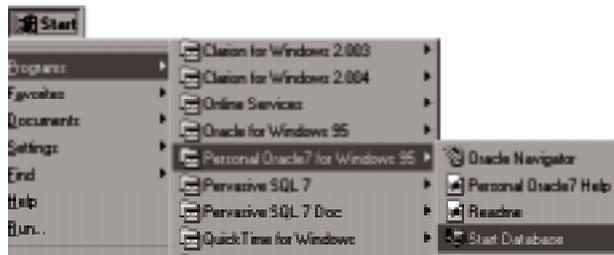
In addition to working through the tutorial, please take a few moments to review the following topics:

- ◆ *Introduction*
  - System Requirements—Software*
  - Registering the Oracle Accelerator*
- ◆ *Using Oracle Accelerator*
  - Overview*
- ◆ *Specifications*
- ◆ *SQL Accelerator Drivers*

## Table Import Wizard—Login Dialog

Clarion's Dictionary Editor Import Wizard lets you import Oracle table definitions into your Clarion Data Dictionary. When you select the Oracle Accelerator from the driver drop-down list, the Import Wizard opens the **Login/Connection** dialog. The **Login/Connection** dialog collects the connection information for the Oracle database.

**Note:** Before you can connect to the SQL database and import table definitions, the database must be started and must be accessible from your computer.



**Note:** Only those indexes directly associated with the table as **CONSTRAINTS** are imported by the Oracle Accelerator driver. If you need more indexes, simply define them manually.

**Note:** If the Oracle database **INDEX** flag is set to **OFF**, the Oracle Accelerator Import Wizard does not import **CONSTRAINTS**.

Fill in the following fields in the **Login/Connection** dialog:

### Host

Select the Oracle host that contains the tables or views to import. If the **Host** list is empty, you may type in the host (a blank host specifies the Oracle 8 Personal database). See your DBA or network administrator for information on how the host is specified. For example, type 2: to connect to the local Personal Oracle (7.2 and earlier) database. X: prefixes an IPX host and TNS: prefixes a TCP/IP host.

**Note:** For Personal Oracle 8, leave the Host field blank.

### Username

Type your Oracle Username. See your server documentation or your DBA for information on applicable Usernames.

### Password

Type your Oracle Password. See your server documentation or your DBA for information on applicable Passwords.

Optionally, you type a complete connect string in the **Username**

field using either of the following syntaxes:

```
username/password@Protocol:dbname
```

or

```
username@Protocol:dbname,password
```

For example type:

```
scott/tiger@2:production1
```

in the **Username** field. Or you may type just your username and the database name in the **Username** field, and type your password in the **Password** field. For example type:

```
scott@2:production1
```

in the **Username** field, then type

```
tiger
```

in the **Password** field. The Import Wizard displays the password as a series of asterisks. See your Oracle documentation for more information on Oracle connect string syntax.



### Filter

Optionally, provide a filter expression to limit the list of tables and views to import. The filter expression queries the ALL\_CATALOG view. For example the filter: OWNER='SCOTT' returns only the tables which have SCOTT as the OWNER. . The filter expression is limited to 1024 characters in length.

**Tip:** The filter is case sensitive, so type your filter value accordingly.

Following is a list of the ALL\_CATALOG column names (and their Clarion datatypes) you can reference in your filter expression. See your Oracle documentation for more information on these columns.

|            |             |
|------------|-------------|
| OWNER      | CSTRING(31) |
| TABLE_NAME | CSTRING(31) |

TABLE\_TYPE

CSTRING(12)

**Next >**

Press this button to open the Import Wizard's **Import List** dialog.

## Table Import Wizard—Import List Dialog

---

When you press the **Next >** button, the Import Wizard opens the **Import List** dialog. The **Import List** dialog lists the importable items.

Highlight the table, view, or synonym whose definition you wish to import, then press the **Finish** button to import. The Import Wizard adds the definition to your Clarion Data Dictionary, then opens the **File Properties** dialog to let you modify the imported definition.

Import additional tables by repeating these steps. After all the items are imported, return to the Dictionary Editor where you can define relationships and delete any columns not used in your Clarion application. See *SQL Accelerators—Define Only the Fields You Use*.

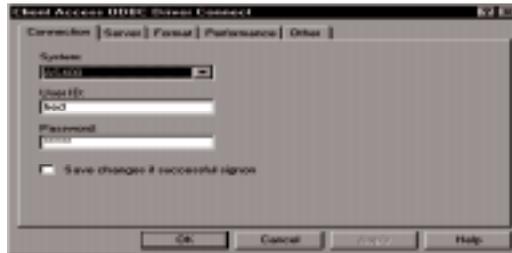
**Tip:** You can use the Enterprise Edition Dictionary Synchronizer to import an entire database, including relationships, in a single pass.

## Performance Considerations

See *SQL Accelerators—Performance Considerations* for more information on performance issues common to all SQL Accelerators, including Oracle Accelerator.

### Automatic Login Dialog

The Oracle Accelerator Login dialog lets the user specify Username, Password and Database.



**Note:** For 16-bit applications with Run-Time Library set to Local, you must add the Clarion \LIBSRC\ORALOON.RSC file to your application's project to use the automatic login dialog. See *Project System* in the *User's Guide* for more information.

In the **Database** drop-down list, select from previously selected Oracle hosts. The list of previously entered databases as well as the last UserID is stored in the Windows registry in the HKEY\_CURRENT\_USER/Software/TopSpeed/Oracle tree as follows:

|                                     |                                 |
|-------------------------------------|---------------------------------|
| /Software/TopSpeed/Oracle/UserID    | the last UserID                 |
| /Software/TopSpeed/Oracle/HostMRU   | the last selected database      |
| /Software/TopSpeed/Oracle/HostCount | number of databases in the list |
| /Software/TopSpeed/Oracle/Host1     | database name                   |
| /Software/TopSpeed/Oracle/Host2     | database name                   |
| /Software/TopSpeed/Oracle/Hostn     | database name                   |

If the **Database** list is empty, you may simply type in the database name. For example, type 2: to connect to the local Personal database. The 2: indicates a local host; X: indicates an IPX host and TNS: indicates a TCP/IP host.

Alternatively, you may also supply a connect string (containing the database name) in the **Username** field. The Oracle connect string syntax is:

```
username/password@Protocol:dbname
or
username@Protocol:dbname,password
```

See your Oracle documentation for more information on Oracle connect string syntax.

If the **Username** field is not long enough, you may continue the entry in the **Password** field, because the Oracle Accelerator driver simply concatenates these fields and forwards their contents to the Oracle server.

## Generating Unique Key Values

---

For many database applications, you will want to automatically generate unique key values for your database records. Both Oracle and Clarion provide tools to help you do this. Oracle provides a Sequence. An Oracle Sequence is simply a sequence number generator. You can select the next number from the Sequence whenever you add a new record. Clarion provides template generated code to automatically increment key values for your database records.

Generally, we recommend using Oracle Sequences whenever possible to generate your unique key values. Sequences are more efficient because you never get a clash, and you need only two (2) database calls to add your new record. With Clarion's template generated code, you will have to make additional database calls if multiple users generate the same sequence number.

### Oracle Sequences

To use Oracle Sequences...

1. Define an Oracle Sequence for the auto incremented key.

See your Oracle documentation:

```
CREATE SEQUENCE CustomerSequence
 INCREMENT BY 1
 START WITH 1
 NOMAXVALUE
 MINVALUE 1
 NOCYCLE
 CACHE 30;
```

2. Declare a Clarion file to receive the sequence number from the Oracle Sequence like this:

```
CustomerSequence FILE, DRIVER('TOPSPEED'), PRE(CUST), CREATE, THREAD
Record RECORD, PRE()
SequenceNo LONG
 END
 END
```

3. Assign the incremented sequence number to your key field by embedding the following in the *WindowManager Method Executable Code Section - PrimeFields* embed point:

```
Access:CustomerSequence.Open
CustomerSequence{Prop:SQL}='Select CustomerSequence.Nextval from dual'
IF ~Access:CustomerSequence.Next()
 Cust:CustNo=SequenceNo
END
```

where *Cust:CustNo* is the label of your auto-incremented key field.

4. Set the embedded code priority to 7500.

### **Clarion AutoNumbered Keys**

To use Clarion autonumbered keys...

1. Open your data dictionary.
2. Highlight the file to change, then press the **Fields/Keys** button.
3. Select the **Keys** tab.
4. Highlight the key to change, then press the **Properties** button.
5. In the **Key Properties** dialog, select the **Attributes** tab, then check the **Auto Number** box.
6. Press the **OK** button to close the **Key Properties** dialog.
7. Close and save your data dictionary.
8. Generate your application using the Clarion or ABC Templates.

The templates generate code to automatically increment the autonumbered key each time a new record is added.

**Note:** Autonumbered keys only work with template generated adds!

## Driver Strings

There are switches or “driver strings” you can set to control the way your application creates, reads, and writes files with a specific driver. Driver strings are simply messages or parameters that are sent to the file driver at run-time to control its behavior. **See *SQL Accelerators in the Application Handbook* for more information on SQL driver strings.**

In addition to the common SQL Accelerator driver strings, Oracle Accelerator supports the following driver strings as well.

### HINT

---

You can tell Oracle Accelerator to generate Oracle hints by using the HINT driver string,

```
DRIVER ('Oracle', '/HINT=hint')
```

by using /HINT in the key name,

```
Key KEY(fieldlist), NAME('name /HINT=[&]hint')
```

with SEND,

```
SendReturn = SEND (file, '/HINT=[&]hint')
```

or with the PROP:Hint property,

```
file{PROP:Hint} = '['&]hint'
HintString = file{PROP:Hint}
```

The square brackets [] above are used to show that the ampersand (&) is optional.

You can either override the base hint or concatenate a hint. If the first character after the = in the KEY hint is an ampersand (&), Oracle Accelerator concatenates the hint onto the FILE hint, otherwise it overrides the FILE hint.

If the first character after the = in the SEND hint is an ampersand (&) or the first character of a hint property is an ampersand, Oracle Accelerator concatenates the hint onto the current hint (the FILE hint and the KEY hint), otherwise it overrides the FILE and KEY hint.

You can also use PROP:Hint to return the hint that is in use (or will be in use if called after a SET, but before the first NEXT or PREVIOUS statement.)

Example:

```
AFile DRIVER('Oracle', '/hint=COST')
AKey KEY(field), NAME('KeyName /HINT=&FIRST_ROWS')
SEND(AFile, '/HINT=FIRST_ROWS')
AFile{PROP:Hint} = 'FIRST_ROWS'
```

## PERSONAL

---

The Personal Oracle 7.1 Server behaves differently than other Oracle servers. When using Personal Oracle 7.1 you should inform Oracle Accelerator so it can tailor the generated SQL especially for Personal Oracle 7.1. For example:

```
DRIVER ('Oracle', '/PERSONAL')
or
SEND (Myfile, '/PERSONAL')
```

**Note:** The /PERSONAL switch is not required for Personal Oracle 7.2 (Personal Oracle for Windows 95).

**Note:** Personal Oracle 8.0 only works with 32-bit programs.

## WHERE

---

In addition to WHERE driver string supported by all the SQL Accelerator drivers, Oracle Accelerator supports the following special WHERE driver string.

### /Where in the FILE Definition

When a FILE declaration references more than one Oracle table, you must tell the Oracle server which columns link the tables together. A /WHERE in the FILE definition specifies the connecting fields between two or more Oracle tables. For example:

```
OrdBrowse FILE, DRIVER('ORACLE', '/WHERE Orders.AccNum=Customer.AccNum'), |
 NAME('Orders, Customer'), PRE(Orb), BINDABLE, THREAD
OrdbKey KEY(-Orb:OrderNumber), NAME('OrdbKey'), PRIMARY
Record RECORD, PRE()
OrderNumber LONG, NAME('OrderNum')
AccountNumber LONG, NAME('Orders.AccNum')
ShipTo STRING(32), NAME('ShipTo')
Name STRING(31), NAME('Name')
 END
 END
```

**Note:** If you use the templates to generate your application, you will not need this technique. The templates automatically generate VIEWS when more than one table is referenced.

## ***Driver Properties***

You can use Clarion's property syntax to query and set certain Oracle Accelerator driver properties. See *SQL Accelerators—SQL Accelerator Properties*.

### **PROP:Hint**

---

See *Driver Strings—HINT*.

## Using Embedded SQL

You can use Clarion's property syntax (PROP:SQL) to send SQL and PL/SQL statements to the Oracle server within the normal execution of your program. For backward compatibility, you can also use SEND to send SQL and PL/SQL statements; however, we recommend using the property syntax.

See *SQL Accelerators in the Application Handbook* for more information on using embedded SQL.

## PL/SQL

---

PL/SQL is Oracle's procedural language extension to Oracle's SQL language. Because PL/SQL statements are managed by the same engine that manages SQL statements, PL/SQL statements may be incorporated into your Clarion programs in the same manner as SQL statements. For example:

```

SQLFile FILE,DRIVER('Oracle'),NAME(SalaryFile)
Record RECORD
SalaryAmount PDECIMAL(5,2),NAME('JOB')
. .
CODE
SqlFile{PROP:SQL} = |
'DECLARE ' &|
'TempPhoneArea clarionclient.PhoneArea%type; ' &|
'CURSOR AreaCursor IS ' &|
'SELECT PhoneArea ' &|
'FROM ClarionClient ' &|
'WHERE PhoneArea = 305; ' &|
'BEGIN ' &|
'OPEN AreaCursor; ' &|
'LOOP ' &|
'FETCH AreaCursor INTO TempPhoneArea; ' &|
'EXIT WHEN AreaCursor%NOTFOUND; ' &|
'UPDATE ClarionClient ' &|
'SET PhoneArea = 954; ' &|
'END LOOP; ' &|
'CLOSE AreaCursor; ' &|
'COMMIT WORK; ' &|
'END;'

```

## Calling a Stored Procedure

---

### NORESULTCALL

For Oracle Accelerator, NORESULTCALL is required for stored procedures that do not return a result set. For example:

```
file{PROP:SQL} = 'NORESULTCALL GrantAccessProcedure'
```

## Supported Attributes and Procedures

### File Attributes Supported

|                                                          |                |
|----------------------------------------------------------|----------------|
| CREATE .....                                             | Y <sup>1</sup> |
| DRIVER( <i>filetype</i> [, <i>driver string</i> ]) ..... | Y              |
| NAME .....                                               | Y              |
| ENCRYPT .....                                            | N              |
| OWNER( <i>password</i> ) .....                           | Y              |
| RECLAIM .....                                            | N              |
| PRE( <i>prefix</i> ) .....                               | Y              |
| BINDABLE .....                                           | Y              |
| THREAD .....                                             | Y              |
| EXTERNAL( <i>member</i> ) .....                          | Y              |
| DLL( <i>[flag]</i> ) .....                               | Y              |
| OEM .....                                                | Y <sup>2</sup> |

### File Structures Supported

|              |   |
|--------------|---|
| INDEX .....  | Y |
| KEY .....    | Y |
| MEMO .....   | N |
| BLOB .....   | N |
| RECORD ..... | Y |

### Index, Key, Memo Attributes Supported

|                             |   |
|-----------------------------|---|
| BINARY .....                | N |
| DUP .....                   | Y |
| NOCASE .....                | Y |
| OPT .....                   | Y |
| PRIMARY .....               | Y |
| NAME .....                  | Y |
| Ascending Components .....  | Y |
| Descending Components ..... | Y |
| Mixed Components .....      | Y |

### Field Attributes Supported

|            |   |
|------------|---|
| DIM .....  | N |
| OVER ..... | Y |
| NAME ..... | Y |

### File Procedures Supported

|                                                 |                 |
|-------------------------------------------------|-----------------|
| BOF( <i>file</i> ) .....                        | N               |
| BUFFER( <i>file</i> ) .....                     | Y               |
| BUILD( <i>file</i> ) .....                      | Y               |
| BUILD( <i>key</i> ) .....                       | Y               |
| BUILD( <i>index</i> ) .....                     | Y <sup>3</sup>  |
| BUILD( <i>index, components</i> ) .....         | Y <sup>3</sup>  |
| BUILD( <i>index, components, filter</i> ) ..... | N               |
| BYTES( <i>file</i> ) .....                      | Y <sup>10</sup> |
| CLOSE( <i>file</i> ) .....                      | Y               |
| COPY( <i>file, new file</i> ) .....             | N               |
| CREATE( <i>file</i> ) .....                     | Y <sup>1</sup>  |
| DUPLICATE( <i>file</i> ) .....                  | Y               |
| DUPLICATE( <i>key</i> ) .....                   | Y               |
| EMPTY( <i>file</i> ) .....                      | Y               |
| EOF( <i>file</i> ) .....                        | N               |
| FLUSH( <i>file</i> ) .....                      | N               |
| LOCK( <i>file</i> ) .....                       | N               |
| NAME( <i>label</i> ) .....                      | Y               |
| OPEN( <i>file, access mode</i> ) .....          | Y               |

|                                         |                 |
|-----------------------------------------|-----------------|
| PACK( <i>file</i> ) .....               | N               |
| POINTER( <i>file</i> ) .....            | N               |
| POINTER( <i>key</i> ) .....             | N               |
| POSITION( <i>file</i> ) .....           | Y <sup>13</sup> |
| POSITION( <i>key</i> ) .....            | Y <sup>11</sup> |
| RECORDS( <i>file</i> ) .....            | Y               |
| RECORDS( <i>key</i> ) .....             | Y <sup>12</sup> |
| REMOVE( <i>file</i> ) .....             | Y               |
| RENAME( <i>file, new file</i> ) .....   | N               |
| SEND( <i>file, message</i> ) .....      | Y               |
| SHARE( <i>file, access mode</i> ) ..... | Y               |
| STATUS( <i>file</i> ) .....             | Y               |
| STREAM( <i>file</i> ) .....             | N               |
| UNLOCK( <i>file</i> ) .....             | N               |

### Record Access Supported

|                                               |                |
|-----------------------------------------------|----------------|
| ADD( <i>file</i> ) .....                      | Y              |
| ADD( <i>file, length</i> ) .....              | N              |
| APPEND( <i>file</i> ) .....                   | Y <sup>4</sup> |
| APPEND( <i>file, length</i> ) .....           | N              |
| DELETE( <i>file</i> ) .....                   | Y              |
| GET( <i>file, key</i> ) .....                 | Y              |
| GET( <i>file, filepointer</i> ) .....         | Y <sup>5</sup> |
| GET( <i>file, filepointer, length</i> ) ..... | N              |
| GET( <i>key, keypointer</i> ) .....           | N              |
| HOLD( <i>file</i> ) .....                     | Y <sup>6</sup> |
| NEXT( <i>file</i> ) .....                     | Y              |
| NOMEMO( <i>file</i> ) .....                   | N              |
| PREVIOUS( <i>file</i> ) .....                 | Y <sup>7</sup> |
| PUT( <i>file</i> ) .....                      | Y              |
| PUT( <i>file, filepointer</i> ) .....         | N              |
| PUT( <i>file, filepointer, length</i> ) ..... | N              |
| RELEASE( <i>file</i> ) .....                  | N              |
| REGET( <i>file, string</i> ) .....            | Y <sup>8</sup> |
| REGET( <i>key, string</i> ) .....             | Y <sup>8</sup> |
| RESET( <i>file, string</i> ) .....            | N              |
| RESET( <i>key, string</i> ) .....             | Y <sup>9</sup> |
| SET( <i>file</i> ) .....                      | Y              |
| SET( <i>file, key</i> ) .....                 | N              |
| SET( <i>file, filepointer</i> ) .....         | N              |
| SET( <i>key</i> ) .....                       | Y              |
| SET( <i>key, key</i> ) .....                  | Y              |
| SET( <i>key, keypointer</i> ) .....           | N              |
| SET( <i>key, key, keypointer</i> ) .....      | N              |
| SKIP( <i>file, count</i> ) .....              | Y              |
| WATCH( <i>file</i> ) .....                    | Y              |

### Transaction Processing Supported

|                                                 |   |
|-------------------------------------------------|---|
| LOGOUT( <i>timeout, file, ..., file</i> ) ..... | Y |
| COMMIT .....                                    | Y |
| ROLLBACK .....                                  | Y |

### Null Data Processing Supported

|                                  |                 |
|----------------------------------|-----------------|
| NULL( <i>field</i> ) .....       | Y               |
| SETNULL( <i>field</i> ) .....    | Y <sup>13</sup> |
| SETNONNULL( <i>field</i> ) ..... | Y               |

## Notes

---

- 1 `CREATE(file)` does not work for every data type. See *Supported Data Types* for more information.
- 2 Adding the OEM attribute causes the driver to generate calls to `NLSSORT` for string fields in a sort sequence (either key components or `PROP:Order` components).
- 3 The `BUILD(dynamic index)` and `BUILD(index)` statements do not perform any disk action. They only initialize internal Oracle driver structures to track key order access and allow `SELECT` statements to be built when you issue `SET(key)` or `SET(key,key)` statements referencing the *index*.
- 4 The `APPEND` statement behaves identically to the `ADD` statement, that is, keys *are* updated by the `APPEND` statement.
- 5 The `GET(file, filepointer)` statement is unsupported for all values of *filepointer* except *filepointer* = 0. In this case, the record position is cleared and `ERRORCODE 35` is returned.
- 6 Apart from the holding records, the `HOLD` statement has another use. Normally, the driver will not reread the record when you execute a `RESET/NEXT` to the current record. Executing a `HOLD` statement before the `RESET/NEXT` forces the driver to reread the record from disk.
- 7 You can't execute a `PREVIOUS` after a `SET(file)` statement. You can only examine the *file* in a forward order.
- 8 The `REGET` statement only works if you have a unique key defined for the file.
- 9 The `RESET(key,position)/NEXT(file)` statement sequence is optimized to retrieve the record from the driver's internal buffer if the code is resetting to the current record. To force the driver to reread the record from disk, execute a `HOLD` statement before the `RESET/NEXT` sequence. This optimization is not in effect within a transaction frame.
- 10 The `BYTES(file)` function returns the number of records in the file *or* the number of bytes in the last record accessed. Following an `OPEN` statement, the `BYTES` function returns the number of records in the *file*. After the file has been accessed by `GET`, `NEXT`, `ADD`, or `PUT`, the `BYTES` function returns the size of the last *record* accessed.
- 11 The `POSITION(key)` function returns (1 + size of the *key* components + the size of the components of the file's primary key). This formula is true even if the first unique key is the same *key* you are positioning on. If no primary key is defined, then the first unique key is considered the primary key.

If there is no unique key, POSITION(*key*) returns 1 + size of the *key* components. In this case RESET(*key*) will reposition to the first occurrence of the key value, since there is no way of uniquely identifying a record. Therefore, the RESET may position on a different record.

- 12 The RECORDS(*key*) function returns the number of UNIQUE occurrences of the first element of the *key*. This is the same as doing an SQL statement of:

```
SELECT COUNT (DISTINCT key_field1) FROM table
```

- 13 SETNULL(*field*) clears the contents of the field.
- 14 The returned POSITION can only be used with REGET(*file,position*) and only for unique keys.

## Data Types

---

The following table matches Clarion data types to their corresponding Oracle data types.

**Tip:** Generally, you should not have to do any manual matching of data types. Rather, you simply import file definitions from your Oracle database into your Clarion data dictionary. The Oracle Accelerator driver automatically selects the proper data types (see *Importing Oracle Files to a Data Dictionary*).

| <u>Oracle data type</u> | <u>Clarion data type</u>                                                      |
|-------------------------|-------------------------------------------------------------------------------|
| CHAR                    | STRING                                                                        |
| VARCHAR2                | CSTRING                                                                       |
| NUMBER                  | REAL                                                                          |
| NUMBER(n,p)             | PDECIMAL                                                                      |
| NUMBER(n,0)             | BYTE <sup>1</sup> ,SHORT <sup>2</sup> ,USHORT <sup>3</sup> ,LONG <sup>4</sup> |
| LONG                    | STRING+GROUP+SHORT <sup>5</sup>                                               |
| LONG RAW                | STRING+GROUP+SHORT <sup>5</sup>                                               |
| DATE                    | DATE or STRING+DATE+TIME <sup>6</sup>                                         |
| RAW                     | STRING                                                                        |
| ROWID                   | STRING(18) <sup>7</sup>                                                       |

### Data Type Notes:

- <sup>1</sup> CREATE will create a NUMBER(3,0).
- <sup>2</sup> CREATE will create a NUMBER(5,0).
- <sup>3</sup> CREATE will create a NUMBER(5,0).

- 4 CREATE will create a NUMBER(10,0).
- 5 Clarion can access Oracle LONG and LONG RAW data types by using a STRING and a GROUP overlaying the STRING. The GROUP consists of a SHORT and a STRING. The SHORT holding the length of the total data (including the length of the SHORT). For example:

```

Comments STRING(1024),NAME('COMMENTS') !Oracle LONG field
COMMENTS_GROUP GROUP,OVER(Comments)
COMMENTS_LENGTH USHORT
COMMENTS_DATA STRING(1022)
END

```

You cannot use the CREATE statement to create rows of type LONG or LONG RAW.

- 6 Clarion can access Oracle DATE data types by using either a DATE or a STRING with a GROUP overlaying the STRING.

If you use a Clarion DATE field, the TIME component of the field is not readable and is set to 0 when writing the field. You may use a CREATE statement to create the table. For example:

```

StartDate DATE,NAME('START_DATE') !Oracle DATE field

```

If you use a Clarion STRING with an overlaid GROUP, the GROUP consists of a DATE and a TIME field. You may not use a CREATE statement to create the table. For example:

```

OraDate STRING(8),NAME('START_DATE') !Oracle DATE field
StartDate_Group GROUP,OVER(OraDate)
StartDate DATE
StartTime TIME
END

```

**Tip:** Your Clarion application should generally reference the DATE field (StartDate), and should not reference the STRING field (OraDate) or the GROUP field (StartDate\_Group).

**However, if the Oracle date stamp is part of the key, you must include the STRING field (not the DATE field) as a key component in your Clarion data dictionary.**

- 7 Oracle ROWID data types are read and written as a STRING(18) of format BBBB BBBB.RRRR.FFFF. Where B, R, and F are hexadecimal numbers representing block, row, and file number respectively. See your Oracle documentation for more information.

CREATE will not create a ROWID row.

## Future Oracle Releases

TopSpeed's Oracle Accelerator automatically works with Oracle versions 7.0 through 8.03. At runtime, the driver initially tries to load the Oracle 8.03 DLLs. If the 8.03 DLLs are not available, it tries the Oracle 8.0 DLLs, 7.3 DLLs, and so on.

**Note: Personal Oracle 8.0 only works with 32-bit programs.**

If the Oracle Accelerator driver finds an appropriate Oracle DLL, it stores the DLL information for use during subsequent sessions. If the driver finds no Oracle DLL, it posts `ERRORCODE() = 90`, `FILEERRORCODE() = 1` and `FILEERROR() = 'Oracle could not be found,'` so you can detect the condition and correct it, either manually or automatically, by specifying the correct DLL information (for example, Oracle 8.1 DLLs) for use by the driver during subsequent sessions.

**Note: The driver tries to load the Oracle DLLs during program initialization only. If you change DLL information you must restart your program to apply the new values.**

The 16-bit Oracle Accelerator stores the Oracle DLL name in the [CWORACLE] section of WIN.INI in the DLL entry. For example:

```
[CWORACLE]
DLL=ORA8.DLL
```

The 32-bit Oracle Accelerator stores the Oracle DLL name in the Windows registry in the DLL value at `HKEY_LOCAL_MACHINE\SOFTWARE\TopSpeed\Oracle`.

The following 32-bit Clarion code resets the (existing) Oracle DLL information in the Windows registry. That is, this program assumes the Oracle Accelerator driver has already put Oracle DLL information in the registry. The program should be compiled and linked as a 32-bit program.

```
PROGRAM !32-bit

HKEY EQUATE(LONG)
HKEY_LOCAL_MACHINE EQUATE(80000002H)
ALL_ACCESS EQUATE(001F003FH)
REG_SZ EQUATE(1)

MAP
 MODULE('Windows 32-bit API')
 RegOpenKeyExA(HKEY hKey,*CSTRING SubKey,ULONG Reserved, |
 ULONG Security,*HKEY DataSize),LONG,RAW,PASCAL,PROC
 RegSetValueExA(HKEY hKey,*CSTRING ValueName,ULONG Reserved, |
 LONG Type,*CSTRING Details, |
 ULONG DataSize),LONG,RAW,PASCAL,PROC
 END
END

MyKey HKEY
Key CSTRING('SOFTWARE\TopSpeed\Oracle')
Value CSTRING('DLL')
Contents CSTRING('ORA8.DLL')
Err LONG

CODE
Err = RegOpenKeyExA(HKEY_LOCAL_MACHINE,Key,0,ALL_ACCESS,MyKey)
IF Err = 0
 Err = RegSetValueExA(MyKey,Value,0,REG_SZ,Contents,SIZE(Contents))
END
IF Err
 MESSAGE('Oracle Accelerator Registry update failed with error: '&Err)
ELSE
 MESSAGE('Oracle Accelerator Registry update succeeded. '&|
 'Please restart your program.')
```

## Troubleshooting

### Clarion Won't Accept Oracle File Driver

---

Clarion's Dictionary Editor allows you to select the Oracle file driver only if Oracle is installed on your machine. That is, the Oracle DLLs must be installed in a directory in your path before Clarion's Dictionary Editor will recognize the Oracle driver. Attempting to run the Oracle example program will produce error windows that tell you which DLLs are missing.

### Oracle Not Available (-1034)

---

If you receive this error (Oracle error number -1034), make sure the Oracle server is properly installed and is available to the client.

### Unable to allocate memory on user side (-1019)

---



This message may indicate some of the required Oracle .dlls are not installed to a directory in your system path. See *System Requirements—Software*.

### Unable to spawn new ORACLE (-9352)

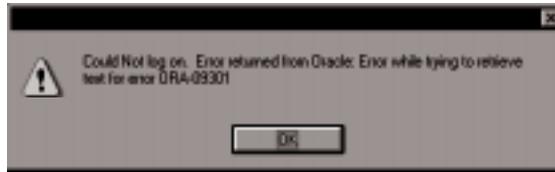
---



This message may indicate the Oracle database is not started. Start the Oracle database and retry.

## Could Not Log On

---



This message may indicate an invalid username, password, or servername. It may also indicate that the Oracle server is not installed or is otherwise not available. Make sure the Oracle server is properly installed and is available to the client.

## Invalid Field Type Descriptor

---

The Clarion error: *Invalid Field Type Descriptor* is generated at runtime if you supply a field name in the field's NAME attribute that does not match any field name in the Oracle table. By turning logging on (see /LOGFILE) you can re-run your program and receive a list of valid field names in the log file. Use the dictionary import facility to import the field descriptions into your Clarion data dictionary to avoid this problem.

## Unexpected End of SQL Command (-921)

---

If you receive this error (Oracle error number -921), make sure that the SQL Select statement selects the same number of columns as the receiving Clarion FILE structure declares.

This error may also occur when a CREATE statement generates incorrect SQL statements when the last field is OVER a previous field. Change the file layout so the last field declared is not declared OVER a previous field.

## File Not Found

---

If you receive this error (or Oracle error number 942—Table or View Does Not Exist), check that the file name you are passing to the Oracle driver is valid. The most common occurrence of this error is when the name of the Oracle table is not correct. If the table is owned by another user, you must explicitly identify the owner as follows:

```
owner.table
```

## Error 47

---

An Error 47 indicates there is a field defined in your Clarion data dictionary that does not exist on the Oracle server. To identify the field, use /LOGFILE.

## Internal Error 02: WSLDIAL

---

Internal error 02: WSLDIAL occurs at runtime for 16-bit applications with Run-Time Library set to Local. The error indicates the program cannot find the resources required for the default login window because they are not included in the project. Typically this happens with a 16-bit application using Run-Time Library set to Local, because resource linking is not as automated for 16-bit applications as for 32-bit applications.



Use the **Project Editor** dialog to add the \CLARION4\LIBSRC\ORALOON.RSC file to your application's **Library, object and resource files**.

To add resource files to your project:

1. From the Application Tree dialog, press the **Project** button.
2. Highlight **Library, object and resource files**, then CLICK on the **Add File** button.
3. Navigate to the CLARION4\LIBSRC folder, then select the resource file (ORALOON.RSC) from the Windows File dialog.
4. Press the **OK** button to return to the **Project Editor** dialog.

## No Interface Driver Connected(-03121)

---

This indicates you have not entered a correct database name. Be sure to enter the correct database name in the **Host** or **Database** field.

## Example Application

The Oracle Accelerator example application is a simple order entry system created as a teaching example. By default, the example application is installed to \Clarion5\Examples\Oracle\Orac.app.

Before running the application, install Oracle Accelerator as described in the *Introduction*, then start your Oracle database.

This section points out the programming techniques used to create the application. Although most procedures use the templates, some “tricks” have been used to optimize performance. In most cases these “tricks” are merely a line or two of embedded source code. In other cases the optimization is accomplished in the data dictionary. These methods are documented in the help for those procedures.

This program also demonstrates different methods to accomplish the same task. By understanding different approaches to problem-solving, you can choose the best method for the job at hand.

## Running the Example Program

---

After you start the program, you should begin by initializing the data files. Under the **Utilities** menu, choose the **Copy Clarion Data...** option or press the **Copy** button to create the Oracle tables. This copies some Clarion data files to Oracle tables on the server to which you connected. When you are finished with the example program, you can use the **Remove Data Files** command to remove the sample tables from the server. You can recreate the tables at any time by pressing the **Copy** button.

## Procedures

---

Most of the procedures in the application are template generated browses and forms. The following procedures are not standard template generated code, but are provided to illustrate various techniques for programming with Clarion and Oracle.

### LogInWindow

Procedure template: WINDOW

This procedure displays an introductory screen to collect Oracle connection information. The connection information is stored in global variables where it is reused for each Oracle table accessed.

Opening the first file takes longer than any other file because the initial connection to Oracle is made at that time. By opening the first file from the first procedure, this time delay is incurred early on in the program.

### **BatchUpdate**

This procedure updates every column in a table with two different methods, and displays how long (in hundredths of a second) it takes to do each update pass. One method uses PROP:SQL to send an SQL statement to the Oracle server to do the batch update (faster). The other method uses the Process template to do the batch update (slower).

### **SQLBuilder**

Procedure template: WINDOW

This procedure builds and executes an SQL select statement with PROP:SQL, then displays the results, demonstrating one way to provide ad hoc queries. Note however, that for the present, the number and data types of the SQL columns selected must match the number and data types of the fields declared in the receiving FILE structure. Therefore, you must declare a series of FILES that match the output of the potential SELECT statements, then you must control the number and sequence of columns selected, and finally, you must direct the output to the matching FILE. A mismatch generates a -921 error code from Oracle.

### **PLSQLFreeForm**

Procedure template: WINDOW

This procedure builds and executes free form PL/SQL code using PROP:SQL. PL/SQL is Oracles procedural language extension for SQL (allows loops, branching, variable declaration and manipulation, etc.)

Use the pre-written PL/SQL block to use a CURSOR to change the area code in the Client file, or write your own free form code. Do not execute code that returns data (that is a SELECT statement). Although this is legal (see the SQLBuilder procedure), this particular procedure is not designed to receive data.

### **PrintClients**

Procedure template: BROWSE

An intermediate procedure using a standard Browse to select a client for which to print invoices. The menu option calling this procedure dictates which Report procedure is called when the user selects a client.

There are two Report procedures, both of which print similar reports; however, different methods are used to generate each report. This demonstrates the flexibility of the Clarion and Oracle tools, plus the performance trade-offs with each method.

**ClarionPrint** uses related files in the file schematic.

**ViewPrint** uses an existing view on the Oracle server.

### **ClarionPrint**

Procedure template: REPORT

This report prints all Invoices for a single client using standard Clarion file definitions. This is the slower of the two report methods used.

### **ViewPrint**

Procedure template: REPORT

This report prints all Invoices for a single client using an SQL VIEW that is stored on the Oracle server.



# 39 - SCALABLE/PERVASIVE.SQL ACCELERATOR

## Overview

### Scalable/Pervasive.SQL Server

---

The newer version of Scalable SQL is called Pervasive.SQL. The following information applies equally to both systems.

### Scalable SQL Driver

---

The Scalable SQL Driver is one of several TopSpeed SQL Accelerator drivers. These SQL Drivers share a common code base and many common features such as TopSpeed's unique, high speed buffering technology, common driver strings, and SQL logging capability. **See *SQL Accelerator Drivers* for information on these common features.**

The Scalable SQL Driver converts standard Clarion file I/O statements and function calls into optimized SQL statements, which it sends to the backend Scalable SQL server for processing. This means you can use the same Clarion code to access both Scalable SQL tables and other file systems such as TopSpeed files. It also means you can use Clarion template generated code with your SQL databases.

All the common behavior of all the SQL Accelerator drivers is documented in the *SQL Drivers* chapter. All behavior specific to the Scalable SQL driver is noted in this chapter.

### SQL Import Wizard—Login Dialog

---

Clarion's Dictionary Editor Import Wizard lets you import Scalable SQL table definitions into your Clarion Data Dictionary. When you select the Scalable SQL Accelerator Driver from the driver drop-down list, the Import Wizard opens the **Login/Connection** dialog. The **Login/Connection** dialog collects the connection information for the Scalable SQL database.

Fill in the following fields in the **Login/Connection** dialog:

#### **Database Name**

Select the Scalable SQL database that contains the tables to import. If the **Database Name** list is empty, you may type in the name.

See your DBA or your server documentation for information on how the database is specified.

### DDF Directory

Press the **Browse** button to select the pathname or directory containing the database DDF files.

### Database Directory

Press the **Browse** button to select the pathname or directory containing the database.

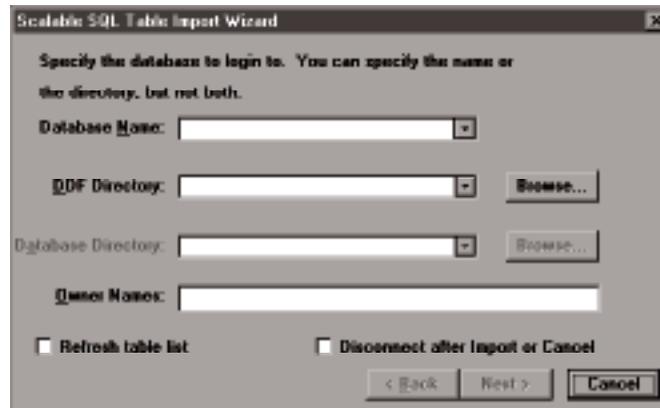
**Note:** You may specify either the Database Name or the DDF directory, but not both.

### User Name

The user login ID for the named database.

### Password

The user password for the named database.



### Owner Names

Optionally, type a comma separated list of names the Scalable SQL driver tries when opening encrypted Btrieve files. If a name contains a comma or space, it must be surrounded by single quotes.

### Refresh table list

Check this box to refresh the list of tables to import when you press the **Next >** button. Clear the box to improve performance when the database is likely to be unchanged between imports.

### Disconnect after Import or Cancel

Check this box to disconnect from the server after importing the (last) definition. Generally, you should clear this box when importing multiple definitions in order to maintain your connection to the server between imports.

### Next >

Press this button to open the Import Wizard's **Import List** dialog.

## SQL Import Wizard—Import List Dialog

---

When you press the **Next >** button, the Import Wizard opens the **Import List** dialog. The **Import List** dialog lists the importable items.

Highlight the table whose definition to import, then press the **Finish** button to import. The Import Wizard adds the definition to your Clarion Data Dictionary, then opens the **File Properties** dialog to let you modify the default definition.

Import additional tables by repeating these steps. After all the items are imported, return to the Dictionary Editor where you can define relationships and delete any columns not used in your Clarion application. See *SQL Accelerator Drivers—Define Only the Fields You Use*.

## Connection Information and Driver Configuration—File Properties

---

Typically, you add Scalable SQL support to your application by importing the table definitions into your Clarion Data Dictionary. The Import Wizard automatically fills in the **File Properties** dialog with default values based on the imported item. However, you can use the **Owner Name** field in the **File Properties** dialog to further configure the way the Scalable SQL Driver accesses the data.

**Tip:** Type an exclamation point (!) followed by the variable name in the **Owner Name** field of the **File Properties** dialog to specify a variable instead of hard coding the connection/configuration information. For example: !GLO:Connection.

Scalable SQL allows some information in addition to the database identification in the **Owner Name** field. This information appears alternatively as:

```
Database[[,[UserID],[Password][,Owners]][;Switches]]
```

or

```
DDF=DDFPath[|Datapath][[,[UserID],[Password][,Owners]][;Switches]]
```

Where:

*Database* is the name of a Scalable SQL database.

*DDFPath* is a path to a set of DDF (Btrieve data dictionary) files.

*Datapath* is the path to the corresponding data files. If the data files reside in multiple directories, then you must specify each directory separated by a semicolon. If *Datapath* is omitted, it defaults to *DDFPath*.

*Owners* is a comma separated list of names (Usernames) to try when opening encrypted Btrieve files. If a name contains a comma or space, it must be surrounded by single quotes.

*Switches* is a semicolon separated list of assignments. Valid switches are:

```
CREATEDDF=[0|1|2]
```

Where *0* creates a new DDF file, *1* replaces the existing DDF file, and *2* removes the existing DDF File.

**Note:** The CREATEDDF switch is provided primarily for use during initial installation to allow you to build new DDF files. You should never use this switch for existing databases.

# ***Performance Considerations***

## **Named Databases**

---

A named database has a logical name that allows users to identify it without knowing its actual location. When you name a database, you associate that name with a particular dictionary directory path and one or more data file paths. When you log in to the SQL Interface using a database name, the SQL Interface uses the name to find the database's dictionary and data files.

A named database lets you do the following:

- Define triggers
- Define primary and foreign keys
- Bind a database
- Suspend a database's integrity constraints
- Apply database security (username and password)

**Tip:** Database names can be defined using the SQL Setup utility that is part of your version of Scalable SQL.

## Driver Strings

There are switches or “driver strings” you can set to control the way your application creates, reads, and writes files with a specific driver. Driver strings are simply messages or parameters that are sent to the file driver at run-time to control its behavior. See *Common Driver Features—Driver Strings* for an overview of these runtime Database Driver switches and parameters.

**Tip:** A forward slash precedes all SQL Accelerator driver strings. The slash allows the driver to distinguish between driver strings and SQL statements sent with SEND.

In addition to the standard SQL Driver Strings, the Scalable SQL Accelerator supports the following Driver Strings:

### LOGONSCREEN

---

```
DRIVER('Scalable', '/LOGONSCREEN = TRUE | FALSE ')
[AutoLogon" =] SEND(file, '/LOGONSCREEN [= TRUE | FALSE]')
```

See PROP:LogonScreen.

### GATHERATOPEN

---

```
DRIVER('Scalable', '/GATHERATOPEN = TRUE | FALSE ')
```

By default the driver delays gathering field information until it is required. However, some backends (like Sybase 11) perform poorly under these conditions. Setting GATHERATOPEN to TRUE forces the driver to gather most of the field information when the file is opened, which avoids a slowdown during program execution.

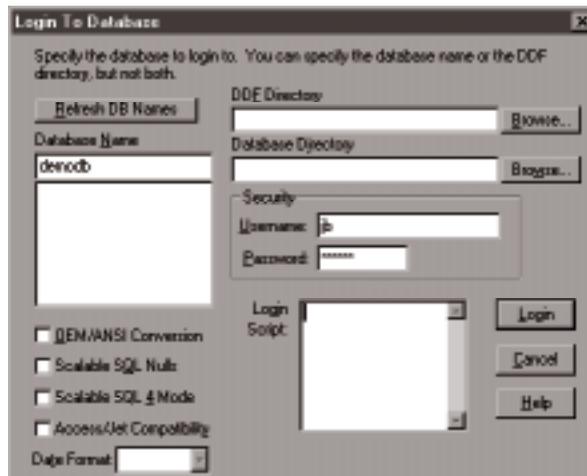
## Driver Properties

You can use Clarion's property syntax to query and set certain Scalable SQL Accelerator properties. In addition to the standard SQL Accelerator properties (see *SQL Accelerators—SQL Accelerator Properties*), the Scalable SQL Accelerator supports the following properties.

### PROP:LogonScreen

PROP:LogonScreen sets or returns the toggle that determines whether the driver automatically prompts for logon information. By default (PROP:LogonScreen=True), the driver does display a logon window if no connect string is supplied. If set to False and there is no connect string, the OPEN(file) fails and FILEERRORCODE() returns '28000.' For example:

```
AFile FILE,DRIVER('Scalable')
!file declaration with no userid and password
END
CODE
AFile{PROP:LogonScreen}=True !enable auto login screen
OPEN(Afile)
```



#### Database Name

Enter the name of a Scalable SQL named database. This option cannot be used with the **DDF Directory** and **Data Directory** options.

#### Database Names List Box

This list box contains the names of databases visible to Scalable SQL from this workstation. This option cannot be used with the **DDF Directory** and **Data Directory** options. Press the **Refresh DB Names** button to reset the list.

**DDF Directory**

The path to the dictionary files (\*.DDF) for your database. This option cannot be used with the **Database Names** option. A new data dictionary will be created (at your option) if one is not located in this directory. The directory must already exist, however.

Press the **Browse** button to select the DDF Directory with the Windows file dialog.

**Database Directory**

The path to the data files for your database, if different from the DDF Directory.

Press the **Browse** button to select the Database Directory with the Windows file dialog.

**User Name**

The user login ID for the named database. This option cannot be used with the **DDF Directory** and **Data Directory** options.

**Password**

The user password for the named database. This option cannot be used with the **DDF Directory** and **Data Directory** options.

**Owner Names**

Enter a list of owner names, up to eight separated by commas, to be used in opening files on connections to this data source.

**Translate OEM characters**

If this check box is checked, character data moving between the ODBC Interface and the underlying database will be translated. Use this option if the database stores character data in the OEM character set, and your application needs to view the data in the character set used by Windows, Windows 95, or Windows NT.

**Scalable SQL Null Handling**

Check this box to apply Scalable SQL-style null handling; clear the box to apply ODBC-style null handling. See your Pervasive documentation for more information on null handling.

**Scalable SQL 4 Mode**

Log into the connection in Scalable SQL 4 mode when checked. When unchecked, login in Scalable SQL 3.x mode even if the underlying engine is Scalable SQL 4 compliant.

**Access/Jet Compatibility**

Check this box to invoke special Access/Jet Compatibility mode that provides additional interoperability with Microsoft Access.

Access is known to have problems with certain data types (notably TIME, DECIMAL, and the various NUMERIC types). These problems show up as #deleted errors and reports that a record has been updated by another user when it has actually not been updated.

When the compatibility mode is turned on, the Interface takes special steps to ensure proper interpretation of these types.

### Date Format

Select the date format. This option is only available when **Scalable SQL 4 Mode** is checked. The options are: mdy, myd, dmy, dym, ymd and ydm. See explanation.

If the date format is set to any value other than the default (mdy), SET DATEFORMAT is called for the specified date format after a successful login to the database. This supports dates using the date format specified for updates and inserts. For example, if Data Format is ymd, a date field can be formatted as in the following SQL statement:

```
Select * from tablename where datecol = '1925/12/25'
```

### Login Script

Enter one or more SQL statements separated by semicolons, to execute immediately after login takes place.

This is useful, for example, to establish global null values when the underlying data was built using null values other than the default. For example:

```
SET BINARYNULL = 255;SET DECIMALNULL = ' '
```

In some cases you may wish to avoid the use of a semi-colon as statement separator. For example, you may wish to use a semi-colon within the SQL statement, or you may need to use the connect string returned from SQLDriverConnect, which uses a semi-colon to separate attribute-value pairs. In these instances, you can begin the script with any non-alphabetic character, and that character will be used as the delimiter. For example:

```
$SET BINARYNULL = 255$SET DECIMALNULL = ' ';'
```

## Supported Attributes and Procedures

### File Attributes Supported

|                                                          |                |
|----------------------------------------------------------|----------------|
| CREATE .....                                             | Y              |
| DRIVER( <i>filetype</i> [, <i>driver string</i> ]) ..... | Y              |
| NAME .....                                               | Y              |
| ENCRYPT .....                                            | N              |
| OWNER( <i>password</i> ) .....                           | Y <sup>1</sup> |
| RECLAIM .....                                            | N              |
| PRE( <i>prefix</i> ) .....                               | Y              |
| BINDABLE .....                                           | Y              |
| THREAD .....                                             | Y              |
| EXTERNAL( <i>member</i> ) .....                          | Y              |
| DLL( <i>flag</i> ) .....                                 | Y              |
| OEM .....                                                | Y <sup>5</sup> |

### File Structures Supported

|              |   |
|--------------|---|
| INDEX .....  | Y |
| KEY .....    | Y |
| MEMO .....   | N |
| BLOB .....   | N |
| RECORD ..... | Y |

### Index, Key, Memo Attributes Supported

|                             |                |
|-----------------------------|----------------|
| BINARY .....                | N <sup>3</sup> |
| DUP .....                   | Y              |
| NOCASE .....                | Y              |
| OPT .....                   | N              |
| PRIMARY .....               | Y              |
| NAME .....                  | Y              |
| Ascending Components .....  | Y              |
| Descending Components ..... | Y              |
| Mixed Components .....      | Y              |

### Field Attributes Supported

|            |   |
|------------|---|
| DIM .....  | N |
| OVER ..... | Y |
| NAME ..... | Y |

### File Procedures Supported

|                                                 |                |
|-------------------------------------------------|----------------|
| BOF( <i>file</i> ) .....                        | N              |
| BUFFER( <i>file</i> ) .....                     | Y              |
| BUILD( <i>file</i> ) .....                      | Y              |
| BUILD( <i>key</i> ) .....                       | Y              |
| BUILD( <i>index</i> ) .....                     | Y <sup>3</sup> |
| BUILD( <i>index, components</i> ) .....         | Y <sup>3</sup> |
| BUILD( <i>index, components, filter</i> ) ..... | N              |
| BYTES( <i>file</i> ) .....                      | Y              |
| CLOSE( <i>file</i> ) .....                      | Y              |
| COPY( <i>file, new file</i> ) .....             | N              |
| CREATE( <i>file</i> ) .....                     | Y              |
| DUPLICATE( <i>file</i> ) .....                  | Y              |
| DUPLICATE( <i>key</i> ) .....                   | Y              |
| EMPTY( <i>file</i> ) .....                      | Y              |
| EOF( <i>file</i> ) .....                        | N              |
| FLUSH( <i>file</i> ) .....                      | N              |
| LOCK( <i>file</i> ) .....                       | N              |
| NAME( <i>label</i> ) .....                      | Y              |
| OPEN( <i>file, access mode</i> ) .....          | Y              |

|                                         |   |
|-----------------------------------------|---|
| PACK( <i>file</i> ) .....               | N |
| POINTER( <i>file</i> ) .....            | N |
| POINTER( <i>key</i> ) .....             | N |
| POSITION( <i>file</i> ) .....           | N |
| POSITION( <i>key</i> ) .....            | Y |
| RECORDS( <i>file</i> ) .....            | Y |
| RECORDS( <i>key</i> ) .....             | Y |
| REMOVE( <i>file</i> ) .....             | Y |
| RENAME( <i>file, new file</i> ) .....   | N |
| SEND( <i>file, message</i> ) .....      | Y |
| SHARE( <i>file, access mode</i> ) ..... | Y |
| STATUS( <i>file</i> ) .....             | Y |
| STREAM( <i>file</i> ) .....             | N |
| UNLOCK( <i>file</i> ) .....             | N |

### Record Access Supported

|                                               |   |
|-----------------------------------------------|---|
| ADD( <i>file</i> ) .....                      | Y |
| ADD( <i>file, length</i> ) .....              | N |
| APPEND( <i>file</i> ) .....                   | Y |
| APPEND( <i>file, length</i> ) .....           | N |
| DELETE( <i>file</i> ) .....                   | Y |
| GET( <i>file, key</i> ) .....                 | Y |
| GET( <i>file, filepointer</i> ) .....         | N |
| GET( <i>file, filepointer, length</i> ) ..... | N |
| GET( <i>key, keypointer</i> ) .....           | N |
| HOLD( <i>file</i> ) .....                     | N |
| NEXT( <i>file</i> ) .....                     | Y |
| NOMEMO( <i>file</i> ) .....                   | N |
| PREVIOUS( <i>file</i> ) .....                 | Y |
| PUT( <i>file</i> ) .....                      | Y |
| PUT( <i>file, filepointer</i> ) .....         | N |
| PUT( <i>file, filepointer, length</i> ) ..... | N |
| RELEASE( <i>file</i> ) .....                  | N |
| REGET( <i>file, string</i> ) .....            | N |
| REGET( <i>key, string</i> ) .....             | Y |
| RESET( <i>file, string</i> ) .....            | N |
| RESET( <i>key, string</i> ) .....             | Y |
| SET( <i>file</i> ) .....                      | Y |
| SET( <i>file, key</i> ) .....                 | N |
| SET( <i>file, filepointer</i> ) .....         | N |
| SET( <i>key</i> ) .....                       | Y |
| SET( <i>key, key</i> ) .....                  | Y |
| SET( <i>key, keypointer</i> ) .....           | N |
| SET( <i>key, key, filepointer</i> ) .....     | N |
| SKIP( <i>file, count</i> ) .....              | Y |
| WATCH( <i>file</i> ) .....                    | Y |

### Transaction Processing Supported<sup>2</sup>

|                                                 |                |
|-------------------------------------------------|----------------|
| LOGOUT( <i>timeout, file, ..., file</i> ) ..... | Y <sup>4</sup> |
| COMMIT .....                                    | Y              |
| ROLLBACK .....                                  | Y              |

### Null Data Processing Supported

|                                  |   |
|----------------------------------|---|
| NULL( <i>field</i> ) .....       | Y |
| SETNULL( <i>field</i> ) .....    | Y |
| SETNONNULL( <i>field</i> ) ..... | Y |

## Notes

---

- 1 We recommend using a variable password that is lengthy and contains special characters because this more effectively hides the password value from anyone looking for it. For example, a password like “dd...#\$...\*&” is much more difficult to “find” than a password like “SALARY.”

**Tip:** Type an exclamation point (!) followed by the variable name in the Owner Name field of the File Properties dialog to specify a variable instead of hard coding the connection information. For example: !GLO:Connection.

- 2 See also *PROP:Logout* in the *Language Reference*.
- 3 BUILD(index) sets internal driver flags to guarantee the driver generates the correct ORDER BY clause. The driver does not call the backend server.
- 4 Whether LOGOUT also LOCKs the table depends on the server’s configuration for transaction processing. See your server documentation.
- 5 When using the Scalable driver with OEM attribute the driver will attempt to load the appropriate translator based on the registered ODBC translators. However, if you are using a 16bit application in a 32bit environment and do not have the 16bit ODBC components of Scalable installed, the driver cannot automatically work out the settings. In this case you need to add to your WIN.INI the following section and value:

```
[CWSCALABLE]
Translator32=<name of translator DLL>
```

The name of this DLL can be obtained from the registry in the HKEY\_LOCAL\_MACHINE\Software\ODBC\ODBCINST.INI\Pervasive Software OEM to ANSI key looking at the Translator value.

Note that you can also set the Translator value in the CWSCALABLE section to override the default ODBC translator in 16bit. In 32bit you can perform the overrides by setting the Translator value in the HKEY\_LOCAL\_MACHINE\Software\TopSpeed\Scalable key.

## Synchronizer Server

Clarion's Enterprise Edition includes the Scalable SQL Synchronizer Server and the Data Dictionary Synchronizer. The Dictionary Synchronizer uses the Synchronizer Server to gather complete information about a Scalable SQL database.

The Scalable SQL Synchronizer Server is one of several used by the Dictionary Synchronizer. All the common behavior of the Synchronizer Servers is documented in the *Enterprise Tools* book. All behavior specific to the Scalable SQL Synchronizer Server is noted in this section.

### Synchronizer Login Dialog

---

Clarion's Dictionary Synchronizer Wizard (Enterprise Edition) lets you import an entire Scalable SQL database definition into your Clarion Data Dictionary in a single pass. During this process, the Synchronizer Wizard opens an Scalable SQL login dialog. This dialog collects the connection information for the Scalable SQL database.

Fill in the following fields in the login dialog:

#### Database

Select the Scalable SQL database that contains the tables or views to import. If the **Database** list is empty, you may type in the name. See your server documentation or your DBA for information on database names. Named databases let you:

Define triggers

Define primary and foreign keys

Bind a database

Suspend a database's integrity constraints

Apply security (username and password)

**Tip:** Database names can be defined using the SQL Setup utility that is part of your version of Scalable SQL.

#### DDF Path

From the drop-down list, select the directory that contains the .DDF files that describe the Scalable SQL database to synchronize. If the **DDF Path** list is empty, you may type in the path or press the **Browse** button to select a .DDF file with the Windows file dialog.

### DataPath

From the drop-down list, select the directory that contains the data files described by the DDF Files, if different than the DDF path. If the **DataPath** list is empty, you may type in the path or press the **Browse** button to select a data file with the Windows file dialog.

### Owners

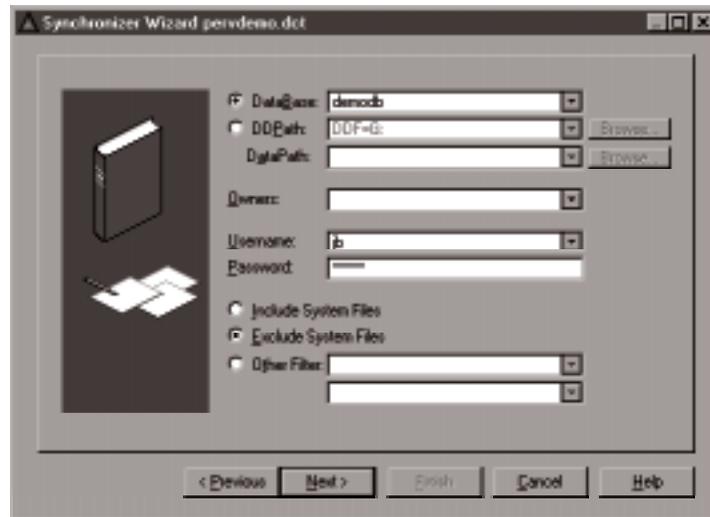
Type a comma separated list of names (up to eight) to try when opening encrypted Btrieve files. If a name contains a comma or space, it must be surrounded by single quotes.

### Username

Type your Scalable SQL Username. See your server documentation or your DBA for information on Usernames.

### Password

Type your Scalable SQL Password. See your server documentation or your DBA for information on Passwords.



### Include System Files

Select this option to include system tables in the list of importable objects.

### Exclude System Files

Select this option to exclude system tables from the list of importable objects.

### Other Filter

Select this option to provide a filter expression to limit the list of tables and views to import. The filter expression is limited to 1024 characters in length.

**Tip:** The filter is case sensitive, so type your filter value accordingly.



# 40 - SQLANYWHERE ACCELERATOR

## Overview

### SQLAnywhere Server

---

For complete information on the SQLAnywhere database system, please review Sybase's SQLAnywhere documentation.

### SQLAnywhere Accelerator

---

The SQLAnywhere Accelerator is one of several TopSpeed SQL Accelerators. These SQL Accelerators share a common code base and many common features such as TopSpeed's unique, high speed buffering technology, common driver strings, and SQL logging capability. **See *SQL Accelerators* for information on these common features.**

The SQLAnywhere Accelerator converts standard Clarion file I/O statements and function calls into optimized SQL statements, which it sends to the backend SQLAnywhere server for processing. This means you can use the same Clarion code to access both SQLAnywhere tables and other file systems such as TopSpeed files. It also means you can use Clarion template generated code with your SQL databases.

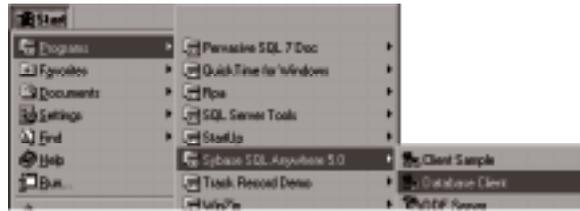
All the common behavior of all the SQL Accelerators is documented in the *SQL Accelerators* chapter. All behavior specific to the SQLAnywhere driver is noted in this chapter.

### Start the SQLAnywhere Client

---

Clarion's Dictionary Synchronizer Wizard (Enterprise Edition) lets you import an entire SQLAnywhere database definition into your Clarion Data Dictionary in a single pass. Before you can connect to the SQLAnywhere database and import table definitions, you must start the database client software.

**Note:** Before you can connect to the SQLAnywhere database and import table definitions, you must start the database client software.



If you have not started the client software, Clarion issues the unable to start database engine message.



## SQL Import Wizard—Login Dialog

Clarion's Dictionary Editor Import Wizard lets you import SQLAnywhere table definitions into your Clarion Data Dictionary. When you select the SQLAnywhere Accelerator from the driver drop-down list, the Import Wizard opens a login dialog. This dialog collects the connection information for the SQLAnywhere database.

Fill in the following fields in the **Login/Connection** dialog:

### Database

Select the SQLAnywhere database that contains the tables to import. If the **Database** list is empty, you may type in the name. See your server documentation or your DBA for information on database names.

### Username

For Standard Security, type your MSSQL Username. For Trusted Security (Integrated NT Security) no Username is required. See your server documentation or your DBA for information on applicable Usernames and security methods.

### Password

For Standard Security, type your MSSQL Password. For Trusted Security (Integrated NT Security) no Password is required. See your server documentation or your DBA for information on applicable Passwords and security methods.



### Filter

Optionally, provide a filter expression to limit the list of tables and views to import. The filter expression queries the SYSCATALOG view. The filter expression is limited to 1024 characters in length.

**Tip:** The filter is case sensitive, so type your filter value accordingly.

Following is a list of the column names (and their Clarion datatypes) you can reference in your filter expression. See your SQLAnywhere documentation for information on these fields.

|             |               |
|-------------|---------------|
| CREATOR     | STRING(128)   |
| TNAME       | STRING(128)   |
| DBSPACENAME | STRING(128)   |
| TABLETYPE   | STRING(10)    |
| NCOLS       | LONG          |
| PRIMARY_KEY | STRING(1)     |
| CHECK       | STRING(32767) |
| REMARKS     | STRING(32767) |

### Disconnect from Server when Import is finished

Check this box to disconnect from the server after importing (or canceling). Generally, you should clear this box when importing multiple definitions in order to maintain your connection to the server between imports.

### Next >

Press this button to open the Import Wizard's **Import List** dialog.

## SQL Import Wizard—Import List Dialog

When you press the **Next >** button, the Import Wizard opens the **Import List** dialog. The **Import List** dialog lists the importable items.

Highlight the table whose definition to import, then press the **Finish** button to import. The Import Wizard adds the definition to your Clarion Data Dictionary, then opens the **File Properties** dialog to let you modify the default definition.

Import additional tables by repeating these steps. After all the items are imported, return to the Dictionary Editor where you can define relationships and delete any columns not used in your Clarion application. See *SQL Accelerators—Define Only the Fields You Use*.

## Connection Information and Driver Configuration—File Properties

Typically, you add SQLAnywhere support to your application by importing the table definitions into your Clarion Data Dictionary. The Import Wizard automatically fills in the **File Properties** dialog with default values based on the imported item.

The OWNER attribute for SQLAnywhere Accelerator takes the format:

```
database,username,password
```

**Tip:** Type an exclamation point (!) followed by a variable name in the Owner Name field of the File Properties dialog to specify a variable connect string rather than hard coding the connect string (OWNER attribute). For example: !GLO:ConnectionString.



## Driver Strings

There are switches or “driver strings” you can set to control the way your application creates, reads, and writes files with a specific driver. Driver strings are simply messages or parameters that are sent to the file driver at run-time to control its behavior. See *Common Driver Features—Driver Strings* for an overview of these runtime Database Driver switches and parameters.

**Tip:** A forward slash precedes all SQL Accelerator driver strings. The slash allows the driver to distinguish between driver strings and SQL statements sent with SEND.

In addition to the standard SQL Driver Strings, the SQLAnywhere Accelerator supports the following Driver Strings:

### LOGONSCREEN

---

```
DRIVER('SQLAnywhere', '/LOGONSCREEN = TRUE | FALSE ')
[AutoLogon" =] SEND(file, '/LOGONSCREEN [= TRUE | FALSE]')
```

See PROP:LogonScreen.

### GATHERATOPEN

---

```
DRIVER('SQLAnywhere', '/GATHERATOPEN = TRUE | FALSE ')
```

By default the driver delays gathering field information until it is required. However, some backends (like Sybase 11) perform poorly under these conditions. Setting GATHERATOPEN to TRUE forces the driver to gather most of the field information when the file is opened, which avoids a slowdown during program execution.

## Driver Properties

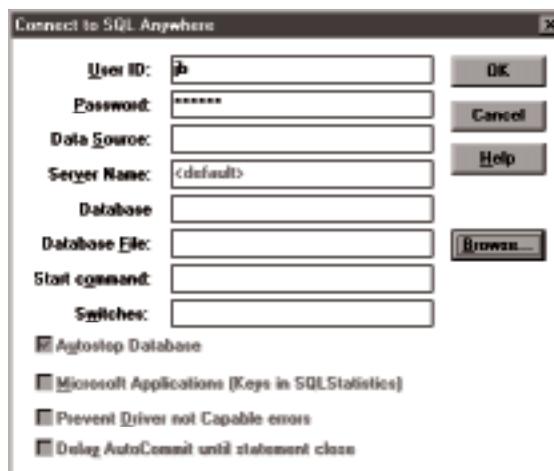
You can use Clarion's property syntax to query and set certain SQLAnywhere Accelerator driver properties. In addition to the standard SQL Accelerator properties (see *SQL Accelerators—SQL Accelerator Properties*), the SQLAnywhere Accelerator supports the following properties.

### PROP:LogonScreen

PROP:LogonScreen sets or returns the toggle that determines whether the driver automatically prompts for logon information. By default (PROP:LogonScreen=True), the driver does display a logon window if no connect string is supplied. If set to False and there is no connect string, the OPEN(file) fails and FILEERRORCODE() returns '28000.' For example:

```
AFile FILE,DRIVER('SQLAnywhere')
!file declaration with no userid and password
END
CODE
AFile{PROP:LogonScreen}=True !enable auto login screen
OPEN(Afile)
```

The logon screen is the SQLAnywhere Connect dialog. Consult your SQLAnywhere documentation for more information on this dialog. The end-user's ability to use the connect dialog will depend on the security surrounding the SQLAnywhere database. For example, the end-users may have access rights to a named database (sademo) that they can access with the SQLAnywhere client software, but they may not have access rights to the \*.db files that comprise the database. The SQLAnywhere connect dialog requires \*.db files rather than database name.



## ***Using Embedded SQL***

You can use Clarion's property syntax (PROP:SQL) to send SQL statements to the backend SQL server within the normal execution of your program. See *SQL Accelerators—Using Embedded SQL* for more information.

## **Calling a Stored Procedure**

---

For SQLAnywhere NORESULTCALL is more efficient than CALL.

## Supported Attributes and Procedures

|                                                                 |                  |                                                                 |                              |
|-----------------------------------------------------------------|------------------|-----------------------------------------------------------------|------------------------------|
| <b>File Attributes</b>                                          | <b>Supported</b> |                                                                 |                              |
| CREATE .....                                                    | Y                | PACK( <i>file</i> ) .....                                       | N                            |
| DRIVER( <i>filetype</i> [, <i>driver string</i> ]) .....        | Y                | POINTER( <i>file</i> ) .....                                    | N                            |
| NAME .....                                                      | Y                | POINTER( <i>key</i> ) .....                                     | N                            |
| ENCRYPT .....                                                   | N                | POSITION( <i>file</i> ) .....                                   | N                            |
| OWNER( <i>password</i> ) .....                                  | Y <sup>1</sup>   | POSITION( <i>key</i> ) .....                                    | Y                            |
| RECLAIM .....                                                   | N                | RECORDS( <i>file</i> ) .....                                    | Y                            |
| PRE( <i>prefix</i> ) .....                                      | Y                | RECORDS( <i>key</i> ) .....                                     | Y                            |
| BINDABLE .....                                                  | Y                | REMOVE( <i>file</i> ) .....                                     | Y                            |
| THREAD .....                                                    | Y                | RENAME( <i>file</i> , <i>new file</i> ) .....                   | N                            |
| EXTERNAL( <i>member</i> ) .....                                 | Y                | SEND( <i>file</i> , <i>message</i> ) .....                      | Y                            |
| DLL([ <i>flag</i> ]) .....                                      | Y                | SHARE( <i>file</i> , <i>access mode</i> ) .....                 | Y                            |
| OEM .....                                                       | N                | STATUS( <i>file</i> ) .....                                     | Y                            |
|                                                                 |                  | STREAM( <i>file</i> ) .....                                     | N                            |
|                                                                 |                  | UNLOCK( <i>file</i> ) .....                                     | N                            |
| <b>File Structures</b>                                          | <b>Supported</b> | <b>Record Access</b>                                            | <b>Supported</b>             |
| INDEX .....                                                     | Y                | ADD( <i>file</i> ) .....                                        | Y                            |
| KEY .....                                                       | Y                | ADD( <i>file</i> , <i>length</i> ) .....                        | N                            |
| MEMO .....                                                      | N                | APPEND( <i>file</i> ) .....                                     | Y                            |
| BLOB .....                                                      | N                | APPEND( <i>file</i> , <i>length</i> ) .....                     | N                            |
| RECORD .....                                                    | Y                | DELETE( <i>file</i> ) .....                                     | Y                            |
|                                                                 |                  | GET( <i>file</i> , <i>key</i> ) .....                           | Y                            |
| <b>Index, Key, Memo Attributes Supported</b>                    |                  | GET( <i>file</i> , <i>filepointer</i> ) .....                   | N                            |
| BINARY .....                                                    | N <sup>3</sup>   | GET( <i>file</i> , <i>filepointer</i> , <i>length</i> ) .....   | N                            |
| DUP .....                                                       | Y                | GET( <i>key</i> , <i>keypointer</i> ) .....                     | N                            |
| NOCASE .....                                                    | Y                | HOLD( <i>file</i> ) .....                                       | N                            |
| OPT .....                                                       | N                | NEXT( <i>file</i> ) .....                                       | Y                            |
| PRIMARY .....                                                   | Y                | NOMEMO( <i>file</i> ) .....                                     | N                            |
| NAME .....                                                      | Y                | PREVIOUS( <i>file</i> ) .....                                   | Y                            |
| Ascending Components .....                                      | Y                | PUT( <i>file</i> ) .....                                        | Y                            |
| Descending Components .....                                     | Y                | PUT( <i>file</i> , <i>filepointer</i> ) .....                   | N                            |
| Mixed Components .....                                          | Y                | PUT( <i>file</i> , <i>filepointer</i> , <i>length</i> ) .....   | N                            |
|                                                                 |                  | RELEASE( <i>file</i> ) .....                                    | N                            |
| <b>Field Attributes</b>                                         | <b>Supported</b> | REGET( <i>file</i> , <i>string</i> ) .....                      | N                            |
| DIM .....                                                       | N                | REGET( <i>key</i> , <i>string</i> ) .....                       | Y                            |
| OVER .....                                                      | Y                | RESET( <i>file</i> , <i>string</i> ) .....                      | N                            |
| NAME .....                                                      | Y                | RESET( <i>key</i> , <i>string</i> ) .....                       | Y                            |
|                                                                 |                  | SET( <i>file</i> ) .....                                        | Y                            |
| <b>File Procedures</b>                                          | <b>Supported</b> | SET( <i>file</i> , <i>key</i> ) .....                           | N                            |
| BOF( <i>file</i> ) .....                                        | N                | SET( <i>file</i> , <i>filepointer</i> ) .....                   | N                            |
| BUFFER( <i>file</i> ) .....                                     | Y                | SET( <i>key</i> ) .....                                         | Y                            |
| BUILD( <i>file</i> ) .....                                      | Y                | SET( <i>key</i> , <i>key</i> ) .....                            | Y                            |
| BUILD( <i>key</i> ) .....                                       | Y                | SET( <i>key</i> , <i>keypointer</i> ) .....                     | N                            |
| BUILD( <i>index</i> ) .....                                     | Y <sup>3</sup>   | SET( <i>key</i> , <i>key</i> , <i>filepointer</i> ) .....       | N                            |
| BUILD( <i>index</i> , <i>components</i> ) .....                 | Y <sup>3</sup>   | SKIP( <i>file</i> , <i>count</i> ) .....                        | Y                            |
| BUILD( <i>index</i> , <i>components</i> , <i>filter</i> ) ..... | N                | WATCH( <i>file</i> ) .....                                      | Y                            |
| BYTES( <i>file</i> ) .....                                      | Y                |                                                                 |                              |
| CLOSE( <i>file</i> ) .....                                      | Y                | <b>Transaction Processing</b>                                   | <b>Supported<sup>2</sup></b> |
| COPY( <i>file</i> , <i>new file</i> ) .....                     | N                | LOGOUT( <i>timeout</i> , <i>file</i> , ..., <i>file</i> ) ..... | Y <sup>4</sup>               |
| CREATE( <i>file</i> ) .....                                     | Y                | COMMIT .....                                                    | Y                            |
| DUPLICATE( <i>file</i> ) .....                                  | Y                | ROLLBACK .....                                                  | Y                            |
| DUPLICATE( <i>key</i> ) .....                                   | Y                |                                                                 |                              |
| EMPTY( <i>file</i> ) .....                                      | Y                | <b>Null Data Processing</b>                                     | <b>Supported</b>             |
| EOF( <i>file</i> ) .....                                        | N                | NULL( <i>field</i> ) .....                                      | Y                            |
| FLUSH( <i>file</i> ) .....                                      | N                | SETNULL( <i>field</i> ) .....                                   | Y                            |
| LOCK( <i>file</i> ) .....                                       | N                | SETNONNULL( <i>field</i> ) .....                                | Y                            |
| NAME( <i>label</i> ) .....                                      | Y                |                                                                 |                              |
| OPEN( <i>file</i> , <i>access mode</i> ) .....                  | Y                |                                                                 |                              |

## Notes

---

- 1 We recommend using a variable password that is lengthy and contains special characters because this more effectively hides the password value from anyone looking for it. For example, a password like “dd....#\$...\*&” is much more difficult to “find” than a password like “SALARY.”

**Tip:** To specify a variable instead of the actual password in the Owner Name field of the File Properties dialog, type an exclamation point (!) followed by the variable name. For example: !MyPassword.

- 2 See also *PROP:Logout* in the *Language Reference*.
- 3 BUILD(index) sets internal driver flags to guarantee the driver generates the correct ORDER BY clause. The driver does not call the backend server.
- 4 Whether LOGOUT also LOCKs the table depends on the server’s configuration for transaction processing. See your server documentation.

## Synchronizer Server

Clarion's Enterprise Edition includes the SQLAnywhere Synchronizer Server and the Data Dictionary Synchronizer. The Dictionary Synchronizer uses the Synchronizer Server to gather complete information about an SQLAnywhere database.

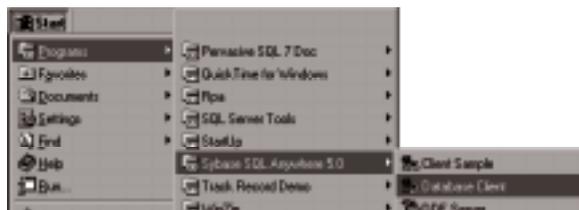
The SQLAnywhere Synchronizer Server is one of several used by the Dictionary Synchronizer. All the common behavior of the Synchronizer Servers is documented in the *Enterprise Tools* book. All behavior specific to the SQLAnywhere Synchronizer Server is noted in this section.

### Start the SQLAnywhere Client

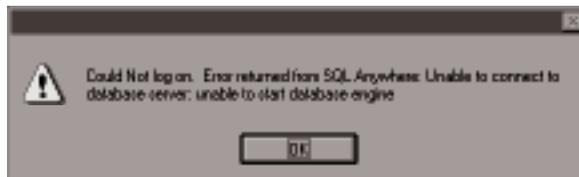
---

Clarion's Dictionary Synchronizer Wizard (Enterprise Edition) lets you import an entire SQLAnywhere database definition into your Clarion Data Dictionary in a single pass. Before you can connect to the SQLAnywhere database and import table definitions, you must start the database client software.

**Note:** Before you can connect to the SQLAnywhere database and import table definitions, you must start the database client software.



If you have not started the client software, Clarion issues the unable to start database engine message.



## Synchronizer Login Dialog

Clarion's Dictionary Synchronizer Wizard (Enterprise Edition) lets you import an entire SQLAnywhere database definition into your Clarion Data Dictionary in a single pass. During this process, the Synchronizer Wizard opens an SQLAnywhere login dialog. This dialog collects the connection information for the SQLAnywhere database.

Fill in the following fields in the login dialog:

### Database

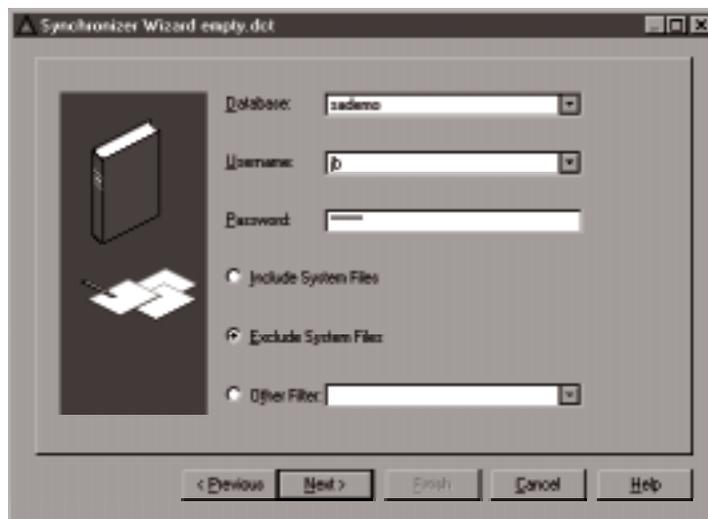
Select the SQLAnywhere database that contains the tables or views to import. If the **Database** list is empty, you may type in the name. See your server documentation or your DBA for information on database names.

### Username

Type your SQLAnywhere Username. See your server documentation or your DBA for information on Usernames.

### Password

Type your SQLAnywhere Password. See your server documentation or your DBA for information on Passwords.



### Include System Files

Select this option to include system tables in the list of importable objects.

### Exclude System Files

Select this option to exclude system tables from the list of importable objects.

**Other Filter**

Select this option to provide a filter expression to limit the list of tables and views to import. The filter expression queries the SYSCATALOG view. The filter expression is limited to 1024 characters in length.

**Tip: The filter is case sensitive, so type your filter value accordingly.**

Following is a list of the column names (and their Clarion datatypes) you can reference in your filter expression. See your SQLAnywhere documentation for information on these fields.

|             |               |
|-------------|---------------|
| CREATOR     | STRING(128)   |
| TNAME       | STRING(128)   |
| DBSPACENAME | STRING(128)   |
| TABLETYPE   | STRING(10)    |
| NCOLS       | LONG          |
| PRIMARY_KEY | STRING(1)     |
| CHECK       | STRING(32767) |
| REMARKS     | STRING(32767) |

# INDEX

## Symbols

|                        |                                                |
|------------------------|------------------------------------------------|
| !                      | 190, 191, 206                                  |
| !!>                    | 175, 180, 207, 405                             |
| #                      | 223, 364                                       |
| #!                     | <b>483</b>                                     |
| #<                     | <b>483</b>                                     |
| #?                     | 475, 476, <b>477</b> , 530, 534, 535, 536      |
| #ABORT                 | <b>473</b>                                     |
| #abort                 | 236                                            |
| #ACCEPT                | <b>402</b> , 540                               |
| #ADD                   | 409, <b>413</b> , 418, 420                     |
| #ALIAS                 | <b>411</b>                                     |
| #and                   | 238                                            |
| #APPEND                | <b>468</b>                                     |
| #APPLICATION           | 364, <b>371</b> , 410                          |
| #AT                    | <b>391</b> , 534, 537                          |
| #ATEND                 | <b>395</b>                                     |
| #ATSTART               | <b>394</b> , 544, 548                          |
| #autocompile           | 233, 234                                       |
| #BOXED                 | 362, <b>445</b> , 526, 544                     |
| #BREAK                 | <b>456</b>                                     |
| #BUTTON                | 362, <b>429</b> , 548                          |
| #CALL                  | <b>460</b>                                     |
| #call                  | 243                                            |
| #CASE                  | 358, <b>457</b>                                |
| #CLASS                 | <b>484</b> , 526                               |
| #CLEAR                 | <b>416</b>                                     |
| #CLOSE                 | <b>465</b>                                     |
| #CODE                  | 358, 359, 364, <b>379</b> , 407, 411, 540      |
| #COMMENT               | <b>485</b>                                     |
| #compile               | 228, 232                                       |
| #CONTEXT               | <b>396</b>                                     |
| #CONTROL               | 358, 359, 365, <b>381</b> , 407, 411, 544      |
| #CREATE                | <b>463</b>                                     |
| #CYCLE                 | <b>456</b>                                     |
| #DEBUG                 | 485                                            |
| #DECLARE               | 197, <b>409</b> , 529, 544, 548                |
| #declare_compiler      | 232, 244                                       |
| #DEFAULT               | 405, 406, <b>408</b>                           |
| #DELETE                | 409, <b>414</b>                                |
| #DELETEALL             | <b>415</b>                                     |
| #deleted TopSpeed ODBC | 676                                            |
| #DISPLAY               | 362, <b>446</b> , 540                          |
| #dolink                | 228, 233                                       |
| #ELSE                  | 452, 457                                       |
| #else                  | 236                                            |
| #ELSIF                 | 452                                            |
| #elsif                 | 236                                            |
| #EMBED                 | 358, 361, <b>388</b> , 475, 476, 477, 494, 529 |
| #EMPTYEMBED            | <b>397</b>                                     |
| #ENABLE                | 362, <b>428</b> , 526                          |
| #ENDAT                 | 391, 394, 395                                  |
| #ENDBOXED              | 445                                            |
| #ENDBUTTON             | 430                                            |
| #ENDCASE               | 457                                            |
| #ENDCONTEXT            | <b>396</b>                                     |
| #ENDDFAULT             | 408                                            |
| #ENDENABLE             | 428                                            |
| #ENDFIELD              | 433                                            |
| #ENDFOR                | 451                                            |
| #ENDGLOBALDATA         | 407                                            |
| #ENDIF                 | 452                                            |
| #endif                 | 236                                            |
| #ENDLOCALDATA          | 407                                            |
| #ENDLOOP               | 454                                            |
| #ENDPREPARE            | 434                                            |
| #ENDREPORTS            | 406                                            |
| #ENDRESTRICT           | 401                                            |
| #ENDSECTION            | <b>469</b>                                     |
| #ENDSHEET              | <b>447</b>                                     |
| #ENDTAB                | <b>448</b>                                     |
| #ENDWINDOWS            | 405                                            |
| #ENDWITH               | 432                                            |
| #EQUATE                | <b>412</b>                                     |
| #ERROR                 | <b>486</b> , 529                               |
| #error                 | 236                                            |
| #exemod                | 244                                            |
| #exists                | 238                                            |
| #expand                | 226, 227, 228                                  |
| #EXPORT                | <b>487</b>                                     |
| #EXTENSION             | 358, 359, 365, <b>385</b> , 407, 411, 548      |
| #FIELD                 | 219, <b>433</b> , 529, 537                     |
| #FIELDS                | 202                                            |
| #file                  | 241                                            |
| #file adderrors        | 228, 242                                       |
| #file append           | 242                                            |
| #file copy             | 241                                            |
| #file delete           | 241                                            |
| #file move             | 242                                            |
| #file redirect         | 242                                            |
| #file touch            | 242                                            |

|                                                 |                              |                                                       |               |
|-------------------------------------------------|------------------------------|-------------------------------------------------------|---------------|
| #FIND .....                                     | 419, 541                     | #pragma call(seg_name => identifier) .....            | 249           |
| #FIX .....                                      | 409, 418, 420                | #pragma call(set JMP => on   off) .....               | 252           |
| #FOR .....                                      | 358, 409, 418, 420, 451      | #pragma call(standard_conv => on   off) .....         | 252           |
| #FREE .....                                     | 409, 417                     | #pragma call(var_arg => on   off) .....               | 251           |
| #GENERATE .....                                 | 473                          | #pragma check(index => on   off) .....                | 257           |
| #GLOBALDATA .....                               | 407                          | #pragma check(nil_ptr => on   off) .....              | 257           |
| #GROUP .....                                    | 364, 376, 409, 459, 526, 529 | #pragma check(overflow => on   off) .....             | 257           |
| #GROUP parameters .....                         | 409                          | #pragma check(range => on   off) .....                | 257           |
| #HELP .....                                     | 487                          | #pragma check(stack => on   off) .....                | 257           |
| #IF .....                                       | 358, 452                     | #pragma classes .....                                 | 246           |
| #if .....                                       | 236                          | #pragma data(c_far_ext => on   off) .....             | 254           |
| #ignore .....                                   | 234                          | #pragma data(class_hierarchy => on   off ) .....      | 255           |
| #IMAGE .....                                    | 446                          | #pragma data(compatible_class => on   off) .....      | 256           |
| #implib .....                                   | 234                          | #pragma data(const_assign => on   off ) .....         | 256           |
| #IMPORT .....                                   | 488                          | #pragma data(const_in_code => on   off) .....         | 255           |
| #INCLUDE .....                                  | 365, 488                     | #pragma data(cpp_compatible_class => on   off ) ..... | 255           |
| #include .....                                  | 243                          | #pragma data(ext_record => on   off ) .....           | 254           |
| #INDENT .....                                   | 458                          | #pragma data(far_ext => on   off ) .....              | 253           |
| #INSERT .....                                   | 364, 459, 526, 529           | #pragma data(near_ptr => on   off) .....              | 254           |
| #INVOKE .....                                   | 461                          | #pragma data(packed => on   off ) .....               | 255           |
| #LINK .....                                     | 202                          | #pragma data(seg_name => identifier) .....            | 253           |
| #link .....                                     | 228, 233                     | #pragma data(stack_size => Number) .....              | 255           |
| #LOCALDATA .....                                | 407, 526, 544                | #pragma data(threshold => Number) .....               | 256           |
| #LOOP .....                                     | 358, 454                     | #pragma data(var_enum_size => on   off) .....         | 255           |
| #MESSAGE .....                                  | 489, 530, 535, 536           | #pragma data(volatile => on   off) .....              | 254           |
| #message .....                                  | 237                          | #pragma data(volatile_variant => on   off ) .....     | 254           |
| #model .....                                    | 228, 231                     | #pragma debug(line_num => on   off) .....             | 263           |
| #MODULE .....                                   | 364, 374                     | #pragma debug(proc_trace => on   off) .....           | 263           |
| #noedit .....                                   | 224                          | #pragma debug(public => on   off) .....               | 263           |
| #not .....                                      | 238                          | #pragma debug(vid => off   min   full) .....          | 262           |
| #OF .....                                       | 457                          | #pragma define(BCD_Arithmetic=> on   off ) .....      | 277           |
| #older .....                                    | 238                          | #pragma define(BCD_Large=> on   off ) .....           | 277           |
| #OPEN .....                                     | 464                          | #pragma define(BCD_ULONG=> on   off ) .....           | 277           |
| #or .....                                       | 237                          | #pragma define(big_code => n) .....                   | 277           |
| #ORIG .....                                     | 202                          | #pragma define(ident=>value) .....                    | 276           |
| #OROF .....                                     | 457                          | #pragma define(init_priority => n) .....              | 277           |
| #POSTEMBED .....                                | 398                          | #pragma define(logical_round => on   off ) .....      | 277           |
| #pragma .....                                   | 229, 231, 246                | #pragma define(maincode => on   off ) .....           | 276           |
| #pragma call(c_conv => on   off) .....          | 248                          | #pragma define(profile => on   off ) .....            | 277           |
| #pragma call(ds_entry => identifier) .....      | 249                          | #pragma define(stack_threshold => size ) .....        | 277           |
| #pragma call(ds_eq_ss => on   off) .....        | 251                          | #pragma define(zero_divide => on   off ) .....        | 276           |
| #pragma call(inline => on   off) .....          | 248                          | #pragma link .....                                    | 232, 233, 274 |
| #pragma call(inline_max => Number) .....        | 252                          | #pragma link_option(decode => on   off ) .....        | 275           |
| #pragma call(near_call => on   off) .....       | 248                          | #pragma link_option(icon => iconname) .....           | 275           |
| #pragma call(o_a_copy => on   off) .....        | 251                          | #pragma link_option(link => <string> ) .....          | 276           |
| #pragma call(o_a_size => on   off) .....        | 250                          | #pragma link_option(map => on   off) .....            | 274           |
| #pragma call(opt_var_arg => on   off) .....     | 252                          | #pragma link_option(shift => num) .....               | 275           |
| #pragma call(reg_param => RegList) .....        | 249                          | #pragma link_option(stub => <filename> ) .....        | 275           |
| #pragma call(reg_return => RegList) .....       | 251                          | #pragma linkfirst .....                               | 234, 274      |
| #pragma call(reg_saved => RegList) .....        | 250                          | #pragma module(implementation => on   off ) .....     | 264           |
| #pragma call(result_optional => on   off) ..... | 251                          | #pragma module(init_code => on   off ) .....          | 264           |
| #pragma call(same_ds => on   off) .....         | 248                          | #pragma module(init_prio => Number) .....             | 264           |

- #pragma module(smart\_link => on | off) ..... 264
- #pragma module(turbo\_comp => on | off ) ..... 264
- #pragma name(prefix => (none | modula | c | os2\_li ..... 258
- #pragma name(prefix => string) ..... 259
- #pragma name(prefix) ..... 258
- #pragma name(upper\_case => on | off) ..... 258
- #pragma optimize(alias => on | off) ..... 261
- #pragma optimize(const => on | off) ..... 260
- #pragma optimize(cse => on | off) ..... 260
- #pragma optimize(jump => on | off) ..... 261
- #pragma optimize(loop => on | off) ..... 261
- #pragma optimize(peep\_hole => on | off) ..... 261
- #pragma optimize(regass => on | off) ..... 261
- #pragma optimize(speed => on | off) ..... 260
- #pragma optimize(stk\_frame => on | off) ..... 261
- #pragma option(bit\_opr => on | off) ..... 266
- #pragma option(incl\_cmt => on | off) ..... 266
- #pragma option(lang\_ext => on | off) ..... 265
- #pragma option(min\_line => on | off) ..... 266
- #pragma option(nest\_cmt => on | off) ..... 266
- #pragma option(pre\_proc => on | off) ..... 266
- #pragma option(uns\_char => on | off) ..... 266
- #pragma project ..... 273
- #pragma restore ..... 274
- #pragma save ..... 273
- #pragma warn(wacc => on | off | err) ..... 272
- #pragma warn(wait => on | off | err) ..... 270
- #pragma warn(wall => on | off | err) ..... 268
- #pragma warn(watr => on | off | err) ..... 269
- #pragma warn(wcic => on | off | err) ..... 273
- #pragma warn(wcid => on | off | err) ..... 270
- #pragma warn(wclt => on | off | err) ..... 271
- #pragma warn(wcne => on | off | err) ..... 270
- #pragma warn(wcor => on | off | err) ..... 271
- #pragma warn(wcrt => on | off | err) ..... 273
- #pragma warn(wdel => on | off | err) ..... 272
- #pragma warn(wdne => on | off | err) ..... 268
- #pragma warn(wdnu => on | off | err) ..... 270
- #pragma warn(wetb => on | off | err) ..... 271
- #pragma warn(wfnd => on | off | err) ..... 269
- #pragma warn(wftn => on | off | err) ..... 270
- #pragma warn(wnid => on | off | err) ..... 272
- #pragma warn(wnre => on | off | err) ..... 269
- #pragma warn(wnrv => on | off | err) ..... 269
- #pragma warn(wntf => on | off | err) ..... 270
- #pragma warn(wovl => on | off | err) ..... 272
- #pragma warn(wovr => on | off | err) ..... 272
- #pragma warn(wpcv => on | off | err) ..... 268
- #pragma warn(wpic => on | off | err) ..... 271
- #pragma warn(wpin => on | off | err) ..... 271
- #pragma warn(wpnd => on | off | err) ..... 269
- #pragma warn(wpnu => on | off | err) ..... 270
- #pragma warn(wprg => on | off | err) ..... 269
- #pragma warn(wral => on | off | err) ..... 271
- #pragma warn(wrpf => on | off | err) ..... 272
- #pragma warn(wsto => on | off | err) ..... 268
- #pragma warn(wtxt => on | off | err) ..... 269
- #pragma warn(wvud => on | off | err) ..... 270
- #pragma warn(wvnu => on | off | err) ..... 272
- #PREEMBED ..... **399**
- #PREPARE ..... **434**
- #PRINT ..... **472**
- #PRIORITY ..... **393**
- #PROCEDURE ..... 359, 364, **375**, 407, 408, 526
- #PROGRAM ..... 364, **373**, 407
- #PROJECT ..... **491**
- #PROMPT ..... 357, 362, 409, **423**, 526
- #prompt ..... 229, 237
- #PROMPT Entry Types ..... 435
- #PROTOTYPE ..... **490**
- #PURGE ..... **416**
- #QUERY ..... **478**
- #READ ..... **466**
- #REDIRECT ..... **467**
- #REJECT ..... **403**, 540
- #RELEASE ..... **475**, 476, 477
- #REMOVE ..... **470**
- #REPLACE ..... **471**
- #REPORTS ..... **406**
- #REQ ..... 382
- #RESTRICT ..... **401**, 540
- #RESUME ..... **476**, 531, 534
- #RETURN ..... **462**, 529
- #RUN ..... **479**
- #run ..... 243
- #RUNDLL ..... **480**
- #rundll ..... 244
- #SECTION ..... **469**
- #SELECT ..... 409, 418, **420**, **421**
- #SEQ ..... 202
- #SERVICE ..... **479**
- #SET ..... 416, **421**
- #set ..... 225, 226
- #SHEET ..... **447**
- #split ..... 227, 228
- #SUSPEND ..... **474**, 475, 477, 530, 533, 535, 536
- #SYSTEM ..... **370**
- #system ..... 229, 230
- #TAB ..... **448**
- #TEMPLATE ..... 364, 368, 369
- #then ..... 236
- #to ..... 232
- #UNFIX ..... **422**
- #UTILITY ..... 190, 358, 365, **378**

|                                          |                              |                                |               |
|------------------------------------------|------------------------------|--------------------------------|---------------|
| #VALIDATE .....                          | 362, 427                     | %ControlTemplate .....         | 518           |
| #WHERE .....                             | 400                          | %ControlTool .....             | 518           |
| #WINDOWS .....                           | 405                          | %ControlToolBar .....          | 518           |
| #WITH .....                              | 432                          | %ControlType .....             | 518           |
| % .....                                  | 212, 213, 223, 225, 360, 364 | %ControlUnsplitStatement ..... | 518           |
| ## .....                                 | 503                          | %ControlUse .....              | 517           |
| %% .....                                 | 503, 550                     | %cpath .....                   | 226, 227      |
| %(expression) .....                      | 504                          | %CreateLocalMap .....          | 507           |
| %@picture@symbol .....                   | 503                          | %devsys .....                  | 227           |
| %[number]symbol .....                    | 503                          | %DictionaryChanged .....       | 506           |
| %  .....                                 | 503, 525                     | %DictionaryFile .....          | 507           |
| %action .....                            | 227                          | %DictionaryToolOptions .....   | 507           |
| %ActiveTemplate .....                    | 516                          | %Driver .....                  | 522           |
| %ActiveTemplateInstance .....            | 517                          | %DriverBinMemo .....           | 522           |
| %ActiveTemplateOwnerInstance .....       | 517                          | %DriverCreate .....            | 522           |
| %ActiveTemplateParentInstance .....      | 517                          | %DriverDescription .....       | 522           |
| %ActiveTemplatePrimaryInstance .....     | 517                          | %DriverDLL .....               | 522           |
| %AliasFile .....                         | 509                          | %DriverEncrypt .....           | 522           |
| %Application .....                       | 506                          | %DriverLIB .....               | 522           |
| %ApplicationDebug .....                  | 506                          | %DriverMaxKeys .....           | 522           |
| %ApplicationLocalLibrary .....           | 506                          | %DriverMemo .....              | 522           |
| %ApplicationTemplate .....               | 506                          | %DriverOpcode .....            | 522           |
| %ApplicationTemplateInstance .....       | 506                          | %DriverOwner .....             | 522           |
| %ApplicationTemplateParentInstance ..... | 506                          | %DriverReclaim .....           | 522           |
| %BytesOutput .....                       | 523                          | %DriverRequired .....          | 522           |
| %cdir .....                              | 226, 227                     | %DriverSQL .....               | 522           |
| %compile_src .....                       | 227                          | %DriverType .....              | 522           |
| %ConditionalGenerate .....               | 523                          | %DriverUniqueKey .....         | 522           |
| %Control .....                           | 517, 536                     | %editfile .....                | 227           |
| %ControlAlert .....                      | 519                          | %EditFilename .....            | 507           |
| %ControlDefaultHeight .....              | 518                          | %EditProcedure .....           | 507           |
| %ControlDefaultWidth .....               | 518                          | %editwin .....                 | 228           |
| %ControlEvent .....                      | 519                          | %EmbedDescription .....        | 523           |
| %ControlField .....                      | 519                          | %EmbedID .....                 | 523           |
| %ControlFieldFormat .....                | 519                          | %EmbedParameters .....         | 523           |
| %ControlFieldHasColor .....              | 519                          | %EOF .....                     | 523           |
| %ControlFieldHasIcon .....               | 519                          | %errors .....                  | 228, 242      |
| %ControlFieldHasLocator .....            | 519                          | %ext .....                     | 226, 227, 228 |
| %ControlFieldHasTree .....               | 519                          | %False .....                   | 523           |
| %ControlFieldHeading .....               | 519                          | %Field .....                   | 509           |
| %ControlFieldPicture .....               | 519                          | %FieldChoices .....            | 511           |
| %ControlFrom .....                       | 519                          | %FieldDerivedFrom .....        | 510           |
| %ControlIndent .....                     | 518                          | %FieldDescription .....        | 510           |
| %ControlInstance .....                   | 518                          | %FieldDimension1 .....         | 510           |
| %ControlMenu .....                       | 518                          | %FieldDimension2 .....         | 510           |
| %ControlMenuBar .....                    | 518                          | %FieldDimension3 .....         | 510           |
| %ControlOriginal .....                   | 518                          | %FieldDimension4 .....         | 510           |
| %ControlParameter .....                  | 518                          | %FieldDisplayChoices .....     | 511           |
| %ControlParent .....                     | 518                          | %FieldDisplayPicture .....     | 510           |
| %ControlParentTab .....                  | 518                          | %FieldFalseValue .....         | 511           |
| %ControlParentType .....                 | 518                          | %FieldFile .....               | 510           |
| %ControlStatement .....                  | 517                          | %FieldFormatWidth .....        | 510           |

|                                 |     |                               |          |
|---------------------------------|-----|-------------------------------|----------|
| %FieldHeader .....              | 510 | %FileReclaim .....            | 508      |
| %FieldHelpID .....              | 510 | %FileRelationType .....       | 513      |
| %FieldID .....                  | 510 | %FileStatement .....          | 508      |
| %FieldIdent .....               | 509 | %FileStruct .....             | 508      |
| %FieldInitial .....             | 510 | %FileStructEnd .....          | 508      |
| %FieldJustIndent .....          | 510 | %FileStructRec .....          | 508      |
| %FieldJustType .....            | 510 | %FileStructRecEnd .....       | 508      |
| %FieldLongDesc .....            | 510 | %FileThreaded .....           | 508      |
| %FieldLookup .....              | 510 | %FileToolOptions .....        | 508      |
| %FieldMemolmage .....           | 510 | %FileType .....               | 508      |
| %FieldMemoSize .....            | 510 | %filetype .....               | 228      |
| %FieldName .....                | 510 | %FileUsage .....              | 508      |
| %FieldPicture .....             | 510 | %FileUserOptions .....        | 508      |
| %FieldPlaces .....              | 510 | %FirstProcedure .....         | 507      |
| %FieldQuickOptions .....        | 511 | %Formula .....                | 520      |
| %FieldRangeHigh .....           | 510 | %FormulaClass .....           | 520      |
| %FieldRangeLow .....            | 510 | %FormulaDescription .....     | 520      |
| %FieldRecordPicture .....       | 510 | %FormulaExpression .....      | 520      |
| %FieldReportControl .....       | 511 | %FormulaExpressionCase .....  | 521      |
| %FieldReportControlHeight ..... | 511 | %FormulaExpressionFalse ..... | 521      |
| %FieldReportControlWidth .....  | 511 | %FormulaExpressionOf .....    | 521      |
| %FieldScreenControl .....       | 511 | %FormulaExpressionTrue .....  | 521      |
| %FieldScreenControlHeight ..... | 511 | %FormulaExpressionType .....  | 521      |
| %FieldScreenControlWidth .....  | 511 | %FormulaInstance .....        | 520      |
| %FieldStatement .....           | 510 | %GlobalData .....             | 507      |
| %FieldStruct .....              | 510 | %GlobalDataLast .....         | 508      |
| %FieldToolOptions .....         | 512 | %GlobalDataLevel .....        | 507      |
| %FieldTrueValue .....           | 511 | %GlobalDataStatement .....    | 507      |
| %FieldType .....                | 510 | %HelpFile .....               | 507      |
| %FieldUserOptions .....         | 511 | %jpicall .....                | 228      |
| %FieldValidation .....          | 511 | %Key .....                    | 509, 512 |
| %FieldValues .....              | 511 | %KeyAuto .....                | 512      |
| %File .....                     | 507 | %KeyDescription .....         | 512      |
| %File32BitOnly .....            | 508 | %KeyDuplicate .....           | 512      |
| %FileBindable .....             | 508 | %KeyExcludeNulls .....        | 512      |
| %FileCreate .....               | 508 | %KeyField .....               | 512      |
| %FileDescription .....          | 507 | %KeyFieldSequence .....       | 512      |
| %FileDriver .....               | 508 | %KeyFile .....                | 512      |
| %FileDriverParameter .....      | 508 | %KeyID .....                  | 512      |
| %FileEncrypt .....              | 508 | %KeyId .....                  | 512      |
| %FileExternal .....             | 508 | %KeyIndex .....               | 512      |
| %FileExternalModule .....       | 508 | %KeyLongDesc .....            | 512      |
| %FileIdent .....                | 507 | %KeyName .....                | 512      |
| %FileKey .....                  | 513 | %KeyNoCase .....              | 512      |
| %FileKeyField .....             | 514 | %KeyOrder .....               | 509, 512 |
| %FileKeyFieldLink .....         | 514 | %KeyOrderAuto .....           | 513      |
| %FileLongDesc .....             | 508 | %KeyOrderDescription .....    | 513      |
| %FileName .....                 | 508 | %KeyOrderDuplicate .....      | 513      |
| %FileOwner .....                | 508 | %KeyOrderExcludeNulls .....   | 513      |
| %FilePrefix .....               | 507 | %KeyOrderFile .....           | 513      |
| %FilePrimaryKey .....           | 508 | %KeyOrderID .....             | 513      |
| %FileQuickOptions .....         | 508 | %KeyOrderIdent .....          | 512      |

|                             |               |                                 |          |
|-----------------------------|---------------|---------------------------------|----------|
| %KeyOrderIndex .....        | 513           | %PrimaryKey .....               | 521      |
| %KeyOrderLongDesc .....     | 513           | %prjname .....                  | 229      |
| %KeyOrderName .....         | 513           | %Procedure .....                | 507, 515 |
| %KeyOrderNoCase .....       | 513           | %ProcedureCalled .....          | 516      |
| %KeyOrderPrimary .....      | 513           | %ProcedureDateChanged .....     | 515      |
| %KeyOrderQuickOptions ..... | 513           | %ProcedureDateCreated .....     | 515      |
| %KeyOrderStatement .....    | 513           | %ProcedureDescription .....     | 516      |
| %KeyOrderStruct .....       | 513           | %ProcedureExported .....        | 516      |
| %KeyOrderToolOptions .....  | 513           | %ProcedureExportsGlobal .....   | 515      |
| %KeyOrderUserOptions .....  | 513           | %ProcedureLanguage .....        | 516      |
| %KeyPrimary .....           | 512           | %ProcedureLongDescription ..... | 516      |
| %KeyQuickOptions .....      | 512           | %ProcedureReadOnly .....        | 515      |
| %KeyStatement .....         | 512           | %ProcedureReturnType .....      | 515      |
| %KeyStruct .....            | 512           | %ProcedureTemplate .....        | 516      |
| %KeyToolOptions .....       | 512           | %ProcedureTimeChanged .....     | 515      |
| %KeyUserOptions .....       | 512           | %ProcedureTimeCreated .....     | 515      |
| %link .....                 | 228           | %ProcedureToDo .....            | 515      |
| %link_arg .....             | 228, 233      | %ProcedureType .....            | 515      |
| %LocalData .....            | 516           | %Program .....                  | 507      |
| %LocalDataDescription ..... | 516           | %ProgramDateChanged .....       | 506      |
| %LocalDataFormatWidth ..... | 516           | %ProgramDateCreated .....       | 506      |
| %LocalDataHeader .....      | 516           | %ProgramExtension .....         | 507      |
| %LocalDataJustIndent .....  | 516           | %ProgramTimeChanged .....       | 506      |
| %LocalDataJustType .....    | 516           | %ProgramTimeCreated .....       | 506      |
| %LocalDataPicture .....     | 516           | %ProjectTarget .....            | 506      |
| %LocalDataStatement .....   | 516           | %Prototype .....                | 515      |
| %main .....                 | 228           | %QuickProcedure .....           | 507      |
| %make .....                 | 228, 232      | %RegistryChanged .....          | 506      |
| %manual_export .....        | 228           | %Relation .....                 | 509, 513 |
| %MenuBarStatement .....     | 517           | %RelationAlias .....            | 513      |
| %model .....                | 228           | %RelationConstraintDelete ..... | 514      |
| %Module .....               | 507, 514      | %RelationConstraintUpdate ..... | 514      |
| %ModuleBase .....           | 514           | %RelationKey .....              | 513      |
| %ModuleChanged .....        | 514           | %RelationKeyField .....         | 514      |
| %ModuleData .....           | 515           | %RelationKeyFieldLink .....     | 514      |
| %ModuleDataStatement .....  | 515           | %RelationPrefix .....           | 513      |
| %ModuleExtension .....      | 514           | %RelationQuickOptions .....     | 514      |
| %ModuleExternal .....       | 514           | %RelationToolOptions .....      | 514      |
| %ModuleInclude .....        | 514           | %RelationUserOptions .....      | 514      |
| %ModuleLanguage .....       | 514           | %remake .....                   | 229      |
| %ModuleProcedure .....      | 515           | %remake_jpi .....               | 229      |
| %ModuleReadOnly .....       | 514           | %reply .....                    | 229, 237 |
| %ModuleTemplate .....       | 514           | %Report .....                   | 519      |
| %name .....                 | 227, 228      | %ReportControl .....            | 519      |
| %Null .....                 | 523           | %ReportControlField .....       | 520      |
| %obj .....                  | 228, 232, 244 | %ReportControlIndent .....      | 520      |
| %odir .....                 | 226, 228      | %ReportControlInstance .....    | 520      |
| %opath .....                | 226, 229      | %ReportControlLabel .....       | 520      |
| %OtherFiles .....           | 521           | %ReportControlOriginal .....    | 520      |
| %pragmastring .....         | 229, 231      | %ReportControlStatement .....   | 520      |
| %Primary .....              | 521           | %ReportControlTemplate .....    | 520      |
| %PrimaryInstance .....      | 521           | %ReportControlType .....        | 520      |

|                                       |                                   |                               |                    |
|---------------------------------------|-----------------------------------|-------------------------------|--------------------|
| %ReportControlUse .....               | 519                               | .CSV, Basic .....             | 567                |
| %ReportStatement .....                | 519                               | .DBD .....                    | 262                |
| %Secondary .....                      | 521                               | .DBF                          |                    |
| %SecondaryCustomJoin .....            | 521                               | Clipper .....                 | 601                |
| %SecondaryCustomText .....            | 521                               | dBaseIII .....                | 615                |
| %SecondaryTo .....                    | 521                               | dBaseIV .....                 | 629                |
| %SecondaryType .....                  | 521                               | FoxPro .....                  | 649                |
| %src .....                            | 229, 232, 244                     | .DEF .....                    | 264                |
| %symbol .....                         | 504                               | .DLL .....                    | 223                |
| %system .....                         | 229                               | .EXE .....                    | 223                |
| %tail .....                           | 226, 229                          | .EXP .....                    | 287                |
| %Target32 .....                       | 506                               | .exp .....                    | 235                |
| %ToolbarStatement .....               | 517                               | .IDX, FoxPro .....            | 649                |
| %True .....                           | 523                               | .INI file .....               | 119                |
| %tsc .....                            | 229                               | .INI file handling code ..... | 530                |
| %tscpp .....                          | 229                               | .ITF .....                    | 264                |
| %tsm2 .....                           | 229                               | .MDX, dBaseIV .....           | 629                |
| %tspas .....                          | 229                               | .MOD .....                    | 304                |
| %ViewFile .....                       | 509                               | .NDX, dBaseIV .....           | 629                |
| %ViewFileField .....                  | 509                               | .OBJ .....                    | 264                |
| %ViewFileFields .....                 | 509                               | .PAS .....                    | 304                |
| %ViewFiles .....                      | 509                               | .PRJ .....                    | 224                |
| %ViewFileStruct .....                 | 509                               | .RSC .....                    | 804                |
| %ViewFileStructEnd .....              | 509                               | .TPE TopSpeed Driver .....    | 678                |
| %ViewFilter .....                     | 508                               | .TPL .....                    | 365, 367           |
| %ViewJoinedTo .....                   | 509                               | .TPR TopSpeed Driver .....    | 678                |
| %ViewPrimary .....                    | 509                               | .TPS TopSpeed Driver .....    | 663                |
| %ViewPrimaryField .....               | 509                               | .TPW .....                    | 365, 367           |
| %ViewPrimaryFields .....              | 509                               | .TR\$, Clarion Driver .....   | 600                |
| %ViewStatement .....                  | 509                               | .TXA .....                    | 488                |
| %ViewStruct .....                     | 508                               | .TXA file format .....        | 408                |
| %ViewStructEnd .....                  | 509                               | .TXA File Sections .....      | 196                |
| %warnings .....                       | 229, 242                          | / .....                       | 232                |
| %Win32 .....                          | 523                               | /Where .....                  | 793                |
| %Window .....                         | 517                               | = .....                       | 238                |
| %WindowEvent .....                    | 517                               | @picture .....                | 212                |
| %WindowStatement .....                | 517                               | [ADDITION] .....              | 219                |
| () .....                              | 238                               | [ALIANSES] .....              | 184                |
| * .....                               | 314, 377, 459, 460, 461, 492, 497 | [APPLICATION] .....           | 196                |
| *.TPL .....                           | 358                               | [CALLS] .....                 | 201                |
| *.TPW .....                           | 358                               | [COMMON] .....                | 197, 199, 201, 204 |
| + (plus sign) in PROP:SQLFilter ..... | 114                               | [DATA] .....                  | 205                |
| -- .....                              | 223                               | [DEFINITION] .....            | 216                |
| -1034 .....                           | 802                               | [DESCRIPTION] .....           | 168                |
| -921 .....                            | 803                               | [DICTIONARY] .....            | 168                |
| -942 .....                            | 803                               | [EMBED] .....                 | 216                |
| .....                                 | 239                               | [FIELDPROMPT] .....           | 212, 219           |
| .APP .....                            | 165, 193, 224                     | [FILES] .....                 | 170, 209           |
| .C .....                              | 304                               | [FORMULA] .....               | 203                |
| .CDX, FoxPro .....                    | 649                               | [INSTANCE] .....              | 209, 219           |
| .CLW .....                            | 304                               | [INSTANCES] .....             | 216                |
| .CPP .....                            | 304                               | [KEY] .....                   | 210                |

|                                  |                              |                                                |               |
|----------------------------------|------------------------------|------------------------------------------------|---------------|
| [LONGDESC] .....                 | 190, 206                     | Clarion .....                                  | 598           |
| [MODULE] .....                   | 200                          | Clipper .....                                  | 606           |
| [OTHERS] .....                   | 210                          | dBaselII .....                                 | 620           |
| [PERSIST] .....                  | 197, 212                     | dBaselV .....                                  | 634           |
| [PRIMARY] .....                  | 209                          | DOS .....                                      | 646           |
| [PROCEDURE] .....                | 200, 201, 217                | FoxPro .....                                   | 654           |
| [PROGRAM] .....                  | 199                          | MSSQL .....                                    | 755           |
| [PROMPTS] .....                  | 212, 220                     | ODBC .....                                     | 776           |
| [QUICKCODE] .....                | 170, 172, 173, 178, 179, 184 | Oracle .....                                   | 796           |
| [RELATIONS] .....                | 185                          | Scalable SQL .....                             | 818, 830      |
| [REPORT] .....                   | 202                          | TopSpeed .....                                 | 667           |
| [REPORTCONTROLS] .....           | 191, 207                     | add file to project .....                      | 491           |
| [SCREENCONTROLS] .....           | 191, 206                     | add to logical procedure tree .....            | 442           |
| [SECONDARY] .....                | 210                          | add to multi-valued symbol .....               | 413           |
| [SOURCE] .....                   | 217                          | add to source file .....                       | 468           |
| [TEMPLATE] .....                 | 217                          | add-in utilities .....                         | 358           |
| [USEROPTION] .....               | 190, 206                     | Adding New Template Sets .....                 | 368           |
| [WINDOW] .....                   | 202                          | adding your own features .....                 | 366           |
| _CDD_ .....                      | 278                          | Addition subsection .....                      | 219           |
| _CLW20_ .....                    | 278                          | Address for local variable not in DGROUP ..... | 272           |
| _CW_ .....                       | 278                          | ADJUST .....                                   | <b>447</b>    |
| _WIDTH32_ .....                  | 278                          | Administrator .....                            | 132           |
| 16-bit CRC .....                 | 97                           | AFTER .....                                    | 380, 382, 386 |
| 32-bit applications .....        | 782                          | ALIAS .....                                    | 312           |
| 3GL .....                        | 303                          | Alias Definition Group .....                   | 184           |
| 3rd Generation Language .....    | 303                          | Aliases .....                                  | 724           |
| <b>A</b>                         |                              | Aliases section .....                          | 184           |
| ABC Templates and SQL .....      | 688                          | aligned target language comments .....         | 483           |
| abort source generation .....    | 473                          | alignment= .....                               | 130           |
| abposh= .....                    | 141, 145                     | ALL_CATALOG .....                              | 787           |
| abposw= .....                    | 141, 145                     | Allocation of registers .....                  | 261           |
| abposx= .....                    | 141, 145                     | ALLOWDETAILS .....                             | 558           |
| abposy= .....                    | 141, 145                     | ALLOWDETAILS, SQL Drivers .....                | 705           |
| abs .....                        | 344                          | AlphaSortEmbeds= .....                         | 136           |
| abstract, general classes .....  | 59                           | Alternate sort orders .....                    | 87            |
| ACCEPT loop .....                | 530                          | ALWAYSQUOTE .....                              |               |
| access .....                     | 351                          | Basic .....                                    | 568           |
| Access 2.0 and ODBC .....        | 778                          | And .....                                      | 238           |
| Access 7.0 and ODBC .....        | 778                          | ANSI .....                                     | 265           |
| Access and ODBC .....            | 762, 778                     | ANSI collating sequence .....                  | 554           |
| Access mode .....                | 93                           | API .....                                      | 327           |
| access to the embed points ..... | 361                          | API Functions .....                            |               |
| Actions dialog .....             | 362, 529, 537                | Prototyping .....                              | 804           |
| Actions popup menu .....         | 361                          | APPEND .....                                   | 735           |
| Actions... dialog .....          | 433                          | ASCII .....                                    | 565           |
| ad hoc queries .....             | 743, 806                     | Basic .....                                    | 572           |
| ADD .....                        | 735                          | Btrieve .....                                  | 583           |
| ASCII .....                      | 565                          | Clarion .....                                  | 598           |
| Basic .....                      | 572                          | Clipper .....                                  | 606           |
| Btrieve .....                    | 583                          | dBaselII .....                                 | 620           |
|                                  |                              | dBaselV .....                                  | 634           |
|                                  |                              | DOS .....                                      | 646           |

- FoxPro ..... 654
  - full key build ..... 665
  - MSSQL ..... 755
  - ODBC ..... 776
  - Oracle ..... 796
  - Scalable SQL ..... 818, 830
  - TopSpeed ..... 667
  - Append ..... 242
  - APPEND Statement ..... 736
  - AppendBuffer ..... 736
  - applets ..... 119
  - APPLICATION ..... 371, 385
  - Application Generator ..... 134
  - Application section ..... 196
  - Application Wizard, SQL Drivers ..... 692
  - ApplicationLoad= ..... 137
  - ApplicationWizard= ..... 134
  - Approximate Record Count, SQL Drivers ..... 696
  - arc ..... 330
  - Array index size check ..... 257
  - ASCII ..... 345
  - ASCII collating sequence ..... 554
  - ASCII File Driver ..... 561
    - Supported File Commands and Functions ..... 565
  - ASCII Files
    - Buffered access behavior ..... 564
    - Data Types ..... 561
    - File Specifications ..... 561
    - QUICKSCAN ..... 564
  - ASCII files
    - Driver Strings ..... 562
    - SEND ..... 562
  - ASCII text ..... 165, 193, 224
  - AskOnCancel= ..... 136
  - AskOnClose= ..... 136
  - AskOnOK= ..... 136
  - ASSIGN ..... 203
  - Assign message= ..... 131
  - assign value to a user-defined symbol ..... 421
  - Assignment in test expression ..... 270
  - asterisk (\*) ..... 377, 459, 460, 461, 492, 497
  - Asterisks ..... 177, 183
  - AT ..... 423, 429, 445, 446
  - atof ..... 343
  - atoi ..... 343
  - atol ..... 343
  - atoul ..... 344
  - attribute string delimiter position ..... 501
  - AUTO ..... 173, 376, 391
  - auto numbered keys ..... 791
  - Auto Populate ..... 130
  - auto-commit mode, SQL Drivers ..... 698
  - Autocompile ..... 274
  - automake= ..... 133
  - autopick= ..... 120
  - autosave= ..... 133
- ## B
- Backup ..... 165, 193
  - Backup Files ..... 239
  - Balloon help ..... 176, 183
  - base class ..... 61
  - Basic File Driver ..... 567
    - Supported File Commands and Functions ..... 572
  - BASIC files
    - Field delimiter ..... 570
    - File Specifications ..... 567
    - Supported Data Types ..... 567
  - Basic Files
    - Buffered access behavior ..... 570
    - QUICKSCAN ..... 570
  - Basic files
    - Driver Strings ..... 568
    - SEND ..... 568
  - BATCH
    - SQL Drivers ..... 701
  - Batch processor ..... 223
  - Batch Updates, SQL Drivers ..... 701
  - BatchChange ..... 743
  - BCD\_Arithmetic ..... 277
  - BCD\_Large ..... 277
  - BCD\_ULONG ..... 277
  - BEFORE ..... 380, 382, 386
  - Before Lookups ..... 532
  - begin conditional source ..... 474
  - behavior (method) ..... 59
  - BFLOAT4 ..... 306
  - BFLOAT8 ..... 306
  - big\_code ..... 277
  - BINARY ..... 735
    - ASCII ..... 565
    - Basic ..... 572
    - Btrieve ..... 583
    - Clarion ..... 598
    - Clipper ..... 606
    - dBaseIII ..... 620
    - dBaseIV ..... 634
    - DOS ..... 646
    - FoxPro ..... 654
    - MSSQL ..... 755
    - ODBC ..... 776
    - Oracle ..... 796
    - Scalable SQL ..... 818, 830

|                             |          |                                             |          |
|-----------------------------|----------|---------------------------------------------|----------|
| TopSpeed .....              | 667      | TopSpeed .....                              | 667      |
| BINDABLE .....              | 726, 735 | BOOLEAN .....                               | 176, 182 |
| ASCII .....                 | 565      | Boolean expressions .....                   | 237      |
| Basic .....                 | 572      | break out of a loop .....                   | 456      |
| Btrieve .....               | 583      | BrowseBox                                   |          |
| Clarion .....               | 598      | duplicated records .....                    | 698      |
| Clipper .....               | 606      | BrowseBox will not scroll .....             | 770      |
| dBaseIII .....              | 620      | Btrieve                                     |          |
| dBaseIV .....               | 634      | ACS .....                                   | 578      |
| DOS .....                   | 646      | ALLOWREAD .....                             | 578      |
| FoxPro .....                | 654      | alternate collating sequence .....          | 578      |
| MSSQL .....                 | 755      | APPENDBUFFER .....                          | 578      |
| ODBC .....                  | 776      | BALANCEKEYS .....                           | 579      |
| Oracle .....                | 796      | Client/Server .....                         | 586      |
| Scalable SQL .....          | 818, 830 | Collating Sequence .....                    | 587      |
| TopSpeed .....              | 667      | COMPRESS .....                              | 579      |
| BINDCONSTANTS               |          | Driver Properties .....                     | 582      |
| ODBC .....                  | 770      | Driver Strings .....                        | 578      |
| bit shift .....             | 331      | File Sharing .....                          | 587      |
| Bitwise operations .....    | 266      | FREESPACE .....                             | 579      |
| BLOB .....                  | 735      | KEY Definitions .....                       | 588      |
| ASCII .....                 | 565      | Keys and Indexes .....                      | 586      |
| Basic .....                 | 572      | LACS .....                                  | 579      |
| Btrieve .....               | 583      | local alternate collating sequence .....    | 579      |
| Clarion .....               | 598      | LVAR .....                                  | 580      |
| Clipper .....               | 606      | MEMO .....                                  | 580      |
| dBaseIII .....              | 620      | NOTE .....                                  | 580      |
| dBaseIV .....               | 634      | Page Size .....                             | 586      |
| DOS .....                   | 646      | PAGESIZE .....                              | 581      |
| FoxPro .....                | 654      | PREALLOCATE .....                           | 581      |
| MSSQL .....                 | 755      | Record Lengths .....                        | 586      |
| ODBC .....                  | 776      | Record Pointers .....                       | 587      |
| Oracle .....                | 796      | TRUNCATE .....                              | 581      |
| Scalable SQL .....          | 818, 830 | Btrieve File Driver                         |          |
| TopSpeed .....              | 663, 667 | Supported File Commands and Functions ..... | 583      |
| BLOB Definition .....       | 178      | Btrieve files .....                         | 575      |
| Blob Definition Group ..... | 178      | File Specifications .....                   | 577      |
| Blueprint .....             | 166, 194 | Supported Data Types .....                  | 576      |
| BOF .....                   | 735      | Btrieve keys                                |          |
| ASCII .....                 | 565      | ANYNULL .....                               | 588      |
| Basic .....                 | 572      | AUTOINCREMENT .....                         | 588      |
| Btrieve .....               | 583      | MODIFIABLE .....                            | 588      |
| Clarion .....               | 598      | REPEATINGDUPLICATE .....                    | 588      |
| Clipper .....               | 606      | Btrieve operations, calling .....           | 582      |
| dBaseIII .....              | 620      | BUFFER .....                                | 735      |
| dBaseIV .....               | 634      | ASCII .....                                 | 565      |
| DOS .....                   | 646      | Basic .....                                 | 572      |
| FoxPro .....                | 654      | Btrieve .....                               | 583      |
| MSSQL .....                 | 755      | Clarion .....                               | 598      |
| ODBC .....                  | 776      | Clipper .....                               | 606      |
| Oracle .....                | 796      | dBaseIII .....                              | 620      |
| Scalable SQL .....          | 818, 830 | dBaseIV .....                               | 634      |

- DOS ..... 646
- FoxPro ..... 654
- MSSQL ..... 755
- ODBC ..... 776
- Oracle ..... 796
- Scalable SQL ..... 818, 830
- TopSpeed ..... 667
- BUFFER statement ..... 111
- Buffered access behavior
  - ASCII ..... 564
  - Basic ..... 570
  - DOS ..... 645
- BUFFERS
  - Clipper ..... 604
  - dBaseIII ..... 618
  - dBaseIV ..... 632
  - FoxPro ..... 652
- BUILD ..... 88, 735
  - ASCII ..... 565
  - Basic ..... 572
  - Btrieve ..... 583
  - Clarion ..... 598
  - Clipper ..... 606
  - dBaseIII ..... 620
  - dBaseIV ..... 634
  - DOS ..... 646
  - FoxPro ..... 654
  - full key build ..... 665
  - MSSQL ..... 755
  - ODBC ..... 776
  - Oracle ..... 796
  - Scalable SQL ..... 818, 830
  - TopSpeed ..... 667
- BUILD Statement ..... 736
- Built-in Symbols ..... 357, 505, 506
- built-in symbols ..... 503
- Built-in Template Functions ..... 493
- built-in template functions ..... 412, 421
- business rules ..... 109
- BY ..... 454
- By address ..... 314
- By value ..... 314
- BYTE ..... 306
- BYTES ..... 735
  - ASCII ..... 565
  - Basic ..... 572
  - Btrieve ..... 583
  - Clarion ..... 598
  - Clipper ..... 606
  - dBaseIII ..... 620
  - dBaseIV ..... 634
  - DOS ..... 646
  - FoxPro ..... 654
  - MSSQL ..... 755
  - ODBC ..... 776
  - Oracle ..... 796
  - Scalable SQL ..... 818, 830
  - TopSpeed ..... 667
- C ..... 245, 303
- C name mangling ..... 258
- C++ ..... 303, 327, 329
- C++ Class Libraries ..... 322
- C/C++
  - Data Type Equivalents ..... 307
  - Interface considerations ..... 324
- C4ORA.DLL ..... 784
- CALL
  - Built-in Template function ..... **492**
  - SQL Drivers ..... 702
- Call #GROUP as a function ..... 492
- Call #pragmas ..... 247
- call Color dialog ..... **435**
- call listbox formatter ..... 439
- Call named #GROUP as a function ..... 497
- call Open File dialog ..... **441**
- call Save File dialog ..... **443**
- Calling 3GL Functions ..... 314
- Calling a Stored Procedure, SQLAnywher ..... 829
- Calling C procedures ..... 251
- Calling convention ..... 247
- Calling Conventions ..... 314, 318
- Calling stored procedures, ODBC ..... 771
- Calling TopSpeed C ..... 252
- CAPS ..... 177, 183
- Capture errors ..... 242
- Cardinals ..... 266
- CASCADE ..... 186, 187
- Cascade the action ..... 83
- CASE ..... 177, 183
- Case as significant when linking ..... 274
- CASE EVENT() ..... 530, 534, 537
- CASE FIELD() ..... 531, 535, 536
- CDX, FoxPro ..... 649
- center ..... 339
- change source file ..... 467
- changes to the template code files ..... 367
- CHAR ..... 732
- character conversion functions ..... 346
- chdir ..... 353
- CHECK ..... 425, **435**
- check boxes ..... 360, 362

|                                                |            |                                             |                    |
|------------------------------------------------|------------|---------------------------------------------|--------------------|
| Check for other user's changes .....           | 96         | CLEAR .....                                 | 186, 187, 428, 445 |
| Checksum .....                                 | 96         | clear single-valued symbol .....            | 416                |
| chmod .....                                    | 351        | Client Access /400 Configuration .....      | 719, 720           |
| CHOICE .....                                   | <b>424</b> | Client workstations .....                   | 105                |
| chrmp .....                                    | 347        | Client-Server, ODBC .....                   | 762                |
| CLABYTE .....                                  | 307        | Client/Server computing .....               | 105                |
| CLADATE .....                                  | 307        | CLIP .....                                  | 339                |
| CLALONG .....                                  | 307        | ASCII .....                                 | 562                |
| CLAREAL .....                                  | 307        | Clipper File Driver                         |                    |
| Clarion DDE Errors .....                       | 163        | Supported File Commands and Functions ..... | 606                |
| Clarion Development Environment .....          | 119        | Clipper files .....                         | 601                |
| Clarion Driver                                 |            | Data Types .....                            | 601                |
| Driver Strings .....                           | 595        | File Specifications .....                   | 603                |
| SEND .....                                     | 595        | CLIPSTRINGS                                 |                    |
| Clarion File Driver                            |            | ODBC .....                                  | 770                |
| Supported File Commands and Functions .....    | 598        | CLOSE .....                                 | 735                |
| Clarion files .....                            | 593        | ASCII .....                                 | 565                |
| DELETED .....                                  | 595        | Basic .....                                 | 572                |
| File Specifications .....                      | 594        | Btrieve .....                               | 583                |
| HELD .....                                     | 595        | Clarion .....                               | 598                |
| IGNORECORRUPTIONS .....                        | 595        | Clipper .....                               | 606                |
| IGNORESTATUS .....                             | 596        | dBaselII .....                              | 620                |
| LOGOUT .....                                   | 600        | dBaselV .....                               | 634                |
| MAINTAINHEADERTIME .....                       | 596        | DOS .....                                   | 646                |
| RECOVER .....                                  | 596        | FoxPro .....                                | 654                |
| Supported Data Types .....                     | 593        | MSSQL .....                                 | 755                |
| Transaction Processing .....                   | 600        | ODBC .....                                  | 776                |
| Clarion FUNCTION .....                         | 315        | Oracle .....                                | 796                |
| Clarion Menu .....                             | 124        | Scalable SQL .....                          | 818, 830           |
| Clarion PROCEDURE .....                        | 315        | TopSpeed .....                              | 667                |
| Clarion Setup Menu .....                       | 125        | close source file .....                     | 465                |
| Clarion templates .....                        | 194        | Codd, E. F. ....                            | 77                 |
| Clarion4.INI .....                             | 119        | CODE .....                                  | 33, 287, 289       |
| ClarionPrint .....                             | 744        | code generation .....                       | 716, 784           |
| ClarionWin .....                               | 151        | Code has no effect .....                    | 270                |
| CLASHORT .....                                 | 307        | code re-use .....                           | 59                 |
| CLASREAL .....                                 | 307        | Code Template .....                         | 539                |
| CLASS .....                                    | 203        | Code template .....                         | 219                |
| Class .....                                    | 205        | code template .....                         | 361, 362, 364, 379 |
| Class definition as function return type ..... | 273        | Code templates .....                        | 359                |
| Class descriptor .....                         | 255        | code templates .....                        | 358, 534, 537      |
| Class hierarchies .....                        | 255        | Collating Sequences .....                   | 554                |
| class hierarchy .....                          | 59         | COLOR .....                                 | <b>425, 435</b>    |
| Class Libraries .....                          | 322        | Color dialog .....                          | <b>435</b>         |
| CLASS Methods .....                            | 62         | column names .....                          | 725                |
| CLASS Properties .....                         | 62         | Column title .....                          | 176, 182           |
| CLASS structure .....                          | 62         | COMMA                                       |                    |
| CLASTRING .....                                | 307        | Basic .....                                 | 568                |
| CLATIME .....                                  | 307        | command buttons .....                       | 360, 362           |
| CLATMP, Clarion Driver .....                   | 600        | command line parameters .....               | 119                |
| CLAULONG .....                                 | 307        | comment block .....                         | 366                |
| CLAUSHORT .....                                | 307        | comment column .....                        | 485                |

- CommentEmbeds= ..... 136
- Comments ..... 190, 223, 266
- COMMIT ..... 730, 735
  - ASCII ..... 565
  - Basic ..... 572
  - Btrieve ..... 583
  - Clarion ..... 598
  - Clipper ..... 606
  - dBaseIII ..... 620
  - dBaseIV ..... 634
  - DOS ..... 646
  - FoxPro ..... 654
  - MSSQL ..... 755
  - ODBC ..... 776
  - Oracle ..... 796
  - Scalable SQL ..... 818, 830
  - SQL Drivers ..... 698
  - TopSpeed ..... 667
- commit conditional source generation ..... 475
- Common subsection ..... 197, 204
- Common Subsections ..... 190, 204
- Compare ..... 339
- Compatibility between compilers ..... 318
- Compatibility with C ..... 255
- Compatibility with Modula-2 ..... 256
- Compatibility with Pascal ..... 256
- COMPILE ..... 276
- Compile
  - Controlling ..... 273, 276
- compile and link a Project or Application ..... 159
- Compile and Link Commands ..... 232
- Compile and Link Options ..... 230
- Compile-and-link ..... 223
- Compiler directives ..... 276
- Compiler Integration ..... 304
- Compiler options ..... 245
- Compiler warnings ..... 266
- Compiling and Linking ..... 230
- COMPONENT ..... 213, 425, **435**
- Composition ..... 71
- composition ..... 65
- concatenate ..... 340
- Concurrency checking ..... 94
- CondGeneration= ..... 134
- Conditional compile ..... 276
- conditional control structures ..... 358
- conditional execution structure ..... 457
- conditional source line ..... 477
- conditionally generate code ..... 452
- conditionally replace source file ..... 471
- Configuration ..... 719, 720
- configuration settings file ..... 119
- Connect String ..... 726
- Connect String, ODBC ..... 767
- Connect String, SQL Drivers ..... 691
- Connect to Clarion ..... 151
- Connect to the Oracle Database ..... 721
- connection information ..... 742
- Constant in code segment requires initialization ..... 272
- Constant is long ..... 271
- Constants ..... 255
  - Frequently used ..... 260
  - Identical ..... 275
- Constraints, SQL Drivers ..... 695
- Constructors ..... 65
- container object ..... 72
- CONTROL ..... 425, **436**
- Control Defaults ..... 143
- Control Panel, ODBC ..... 763
- control prompts ..... 433
- Control Template ..... 543
- Control template ..... 209, 210, 219, 220
- control template ..... 362, 365, 381
- Control templates ..... 359
- control templates ..... 358, 361
- CONTROLS ..... 382, 544
- Convention
  - Calling ..... 314
  - Naming ..... 314
- Conversion may lose significant digits ..... 270
- Conversion of public names ..... 258
- Conversions ..... 343
- COPY ..... 735
  - ASCII ..... 565
  - Basic ..... 572
  - Btrieve ..... 583
  - Clarion ..... 598
  - Clipper ..... 606
  - dBaseIII ..... 620
  - dBaseIV ..... 634
  - DOS ..... 646
  - FoxPro ..... 654
  - MSSQL ..... 755
  - ODBC ..... 776
  - Oracle ..... 796
  - Scalable SQL ..... 818, 830
  - TopSpeed ..... 667
- Copy ..... 241
- Copy Utility TopSpeed Files
  - TopSpeed Files ..... 682
- corrupt data files ..... 556
- Corrupted TopSpeed Files ..... 677
- cosine ..... 330, 332
- Could Not Log On ..... 803

|                                            |               |                                             |               |
|--------------------------------------------|---------------|---------------------------------------------|---------------|
| CRC function .....                         | 97            | Data dictionary .....                       | 196           |
| CREATE .....                               | 735, 736, 785 | Data Dictionary Options .....               | 131           |
| ASCII .....                                | 565           | data entry validation .....                 | 539           |
| Basic .....                                | 572           | data fields in the dictionary .....         | 362           |
| Btrieve .....                              | 583           | Data Independence .....                     | 553           |
| Clarion .....                              | 598           | Data Integrity                              |               |
| Clipper .....                              | 606           | caching .....                               | 556           |
| dBaseIII .....                             | 620           | Data pointers .....                         | 254           |
| dBaseIV .....                              | 634           | data response times .....                   | 108           |
| DOS .....                                  | 646           | Data segment .....                          | 248           |
| FoxPro .....                               | 654           | Data subsection .....                       | 205           |
| MSSQL .....                                | 755           | Data Type Equivalents                       |               |
| ODBC .....                                 | 776           | C/C++ .....                                 | 307           |
| Oracle .....                               | 796, 798, 799 | Modula-2 .....                              | 310           |
| Scalable SQL .....                         | 818, 830      | Pascal .....                                | 312           |
| TopSpeed .....                             | 667           | Data Type Translation .....                 | 306, 315      |
| create source file .....                   | 463           | Data Types .....                            | 306, 732      |
| CREATE statement (SQL) .....               | 113           | Data validation .....                       | 109           |
| CREATED .....                              | 168           | Database Design .....                       | 77            |
| Creating Objects .....                     | 63            | Database Directory, Scalable SQL .....      | 810           |
| CSTRING .....                              | 306, 315, 732 | Database drivers .....                      | 553           |
| CString .....                              | 337           | Database Name, Scalable SQL                 |               |
| CSV, Basic .....                           | 567           | 731, 746, 754, 757, 809, 820, 833           |               |
| CTRLZISEOF                                 |               | Database Name, SQLAnywhere .....            | 824           |
| ASCII .....                                | 562           | Database Recovery Utility TopSpeed .....    | 677           |
| Basic .....                                | 569           | DATE .....                                  | 306, 312, 732 |
| CUI .....                                  | 288, 298      | Oracle .....                                | 798, 799      |
| Current directory .....                    | 196           | date .....                                  | 333, 337      |
| current instance number .....              | 496           | DATE field                                  |               |
| Current Page Number .....                  | 129           | as a key component .....                    | 799           |
| Current System Date .....                  | 129           | Date modified .....                         | 234           |
| Current System Time .....                  | 129           | DATE, Nulls and ODBC .....                  | 772           |
| custom code .....                          | 361           | Date stamp .....                            | 234           |
| custom dialog boxes .....                  | 360           | day .....                                   | 333           |
| customizations .....                       | 359           | DBA tasks .....                             | 785           |
| customize the application .....            | 358           | dBase III File Driver                       |               |
| customize the templates .....              | 358           | Supported File Commands and Functions ..... | 620           |
| Customizing Default Templates .....        | 366           | dBase III files .....                       | 615           |
| CW.TPL .....                               | 365           | Data Types .....                            | 615           |
| CW15.RED .....                             | 196           | File Specifications .....                   | 617           |
| Cw21ProcedureCall= .....                   | 135           | dBase IV .....                              | 629           |
| CWBSY___ Internal Error .....              | 740           | Data Types .....                            | 630           |
| CWC21, Clarion Driver .....                | 600           | File Specifications .....                   | 631           |
| cycle to top of loop .....                 | 456           | Supported Data Types .....                  | 629           |
| Cyclical Redundancy Check (CRC) .....      | 96            | dBaseIV                                     |               |
|                                            |               | Supported File Commands and Functions ..... | 634           |
| <b>D</b>                                   |               | DBF                                         |               |
| DATA .....                                 | 287, 291, 388 | Clipper .....                               | 601           |
| Data declaration section .....             | 33            | dBaseIII .....                              | 615           |
| Data declaration section of a MEMBER ..... | 36            | dBaseIV .....                               | 629           |
| Data Dictionary .....                      | 721           | FoxPro .....                                | 649           |
|                                            |               | DBMS, ODBC .....                            | 762           |

- DDE ..... 149
- DDE Error Messages ..... 162
- DDE identifier for Clarion for Windows ..... 151
- DDE Service Errors ..... 163
- DDE.CLW ..... 150
- DDECLIENT ..... 149, 151
- DDECLUSE ..... 149, 150, 152
- DDEEXECUTE ..... 150, 153, 154, 155, 156, 157, 158, 159, 160, 161
- DDERead ..... 150, 162
- DDESERVER ..... 149
- DDETimeout property ..... 149
- DDF Directory, Scalable SQL ..... 810
- “Deadly Embrace” ..... 103
- DEBUG ..... 297, 300
- Debug File= ..... 135
- DebugGeneration= ..... 134
- Debugging ..... 229, 262
- Debugging File I/O ..... 554
- Debugging Your SQL Application ..... 703
  - ODBC ..... 769
- Debugging ..... 266
- Debugresume= ..... 133
- DECIMAL ..... 306, 732
- Decision Support Systems (DSS) ..... 105
- Declaration has no effect ..... 268
- declarations ..... 327
- declare a page of a #SHEET control ..... 448
- declare a user-defined symbol ..... 409
- declare and assign value to a user-defined symbol ..... 412
- Declare\_compiler ..... 229
- DEFAULT ..... 213, 423
- Default access specifier used for base class ..... 272
- Default Data and Code ..... 405
- Default driver= ..... 131
- default global data declarations ..... 407
- default local data declarations ..... 407
- Default message text ..... 176, 182
- Default naming ..... 264
- default procedure starting point ..... 408
- default procedures ..... 366
- Default prompt ..... 176, 182
- Default report controls ..... 191, 207
- default report structures ..... 406
- Default Size Options ..... 143
- Default tool tip ..... 176, 183
- Default type promotion on parameter ..... 271
- Default window controls ..... 191, 206
- default window structures ..... 405
- default= ..... 128
- Default32bit= ..... 133
- DefaultDictionary= ..... 134
- defaults in the registry ..... 365
- DEFINE ..... 203
- define a formula class ..... 484
- Delay macro substitution ..... 225
- DELETE ..... 186, 735
  - ASCII ..... 565
  - Basic ..... 572
  - Btrieve ..... 583
  - Clarion ..... 598
  - Clipper ..... 606
  - dBaseIII ..... 620
  - dBaseIV ..... 634
  - DOS ..... 646
  - FoxPro ..... 654
  - MSSQL ..... 755
  - ODBC ..... 776
  - Oracle ..... 796
  - Scalable SQL ..... 818, 830
  - TopSpeed ..... 667
- Delete ..... 241
- delete a multi-valued symbol instance ..... 414
- delete a source file ..... 470
- delete all single or multi-valued symbol instances ..... 416
- delete all values from the symbol ..... 416
- delete multiple multi-valued symbol instances ..... 415
- DELETED
  - Clarion ..... 595
  - Clipper ..... 605
  - dBaseIII ..... 619
  - dBaseIV ..... 633
  - FoxPro ..... 653
- delimit conditional source ..... 476
- Deny All ..... 93
- Deny None ..... 93
- Deny Read ..... 93
- Deny Write ..... 93
- DEPEND ..... 214
- Dependent Prompts ..... 213
- dependent symbols ..... 410
- DEPRECATED ..... **389**
- Dereference of NULL pointers ..... 257
- derived class ..... 61
- Derived CLASSES ..... 68
- DESCRIPTION ..... 204, 379, 382, 385, **391**, **393**
- Descriptive text ..... 168, 190, 204, 206
- Destructors ..... 65
- DICTIONARY ..... 196
- Dictionary Properties ..... 168
- Dictionary section ..... 168
- Different const attributes ..... 269
- DIM
  - Oracle ..... 796

|                                             |          |                                      |                 |
|---------------------------------------------|----------|--------------------------------------|-----------------|
| Directory .....                             | 239      | DOUBLE PRECISION .....               | 732             |
| Directory paths .....                       | 240, 241 | downsizing .....                     | 105             |
| disable prompts .....                       | 428      | DRIVER .....                         | 735             |
| DisableField= .....                         | 135      | ASCII .....                          | 565             |
| disambiguate .....                          | 72       | Basic .....                          | 572             |
| Display field description= .....            | 131      | Btrieve .....                        | 583             |
| Display field picture= .....                | 131      | Clarion .....                        | 598             |
| Display field prefix= .....                 | 131      | Clipper .....                        | 606             |
| Display field type= .....                   | 131      | dBaseIII .....                       | 620             |
| Display file description= .....             | 132      | dBaseIV .....                        | 634             |
| Display file driver= .....                  | 131      | DOS .....                            | 646             |
| Display file prefix= .....                  | 132      | FoxPro .....                         | 654             |
| display graphic .....                       | 446      | MSSQL .....                          | 755             |
| Display key attributes= .....               | 131      | ODBC .....                           | 776             |
| Display key description= .....              | 131      | Oracle .....                         | 796             |
| Display key prefix= .....                   | 131      | Scalable SQL .....                   | 818, 830        |
| Display key primary= .....                  | 131      | TopSpeed .....                       | 667             |
| Display key type= .....                     | 131      | Driver .....                         |                 |
| Display key unique= .....                   | 131      | choosing .....                       | 553             |
| display radio buttons .....                 | 442      | Driver Properties .....              |                 |
| display-only prompt .....                   | 446      | Btrieve .....                        | 582             |
| DISPOSE .....                               | 64       | MSSQL .....                          | 731, 753        |
| division by zero .....                      | 276      | Scalable SQL .....                   | 815             |
| DLL .....                                   | 321, 735 | SQLAnywhere .....                    | 828             |
| ASCII .....                                 | 565      | Driver Properties, all drivers ..... | 559             |
| Basic .....                                 | 572      | Driver Strings .....                 |                 |
| Btrieve .....                               | 583      | Btrieve .....                        | 578             |
| Clarion .....                               | 598      | Clipper .....                        | 604             |
| Clipper .....                               | 606      | dBaseIII .....                       | 618             |
| dBaseIII .....                              | 620      | dBaseIV .....                        | 632             |
| dBaseIV .....                               | 634      | FoxPro .....                         | 652             |
| DOS .....                                   | 646      | MSSQL .....                          | 751             |
| FoxPro .....                                | 654      | ODBC .....                           | 770             |
| MSSQL .....                                 | 755      | Oracle .....                         | 792             |
| ODBC .....                                  | 776      | Scalable SQL .....                   | 814             |
| Oracle .....                                | 796      | SQLAnywhere .....                    | 827             |
| Scalable SQL .....                          | 818, 830 | Driver Strings, all drivers .....    | 557             |
| TopSpeed .....                              | 667      | DROP .....                           | 425, <b>436</b> |
| Dll .....                                   | 230      | droplist of items .....              | 436             |
| DOS Access Code .....                       | 93       | DSS .....                            | 105             |
| DOS DLLs .....                              | 244      | DUP .....                            | 735             |
| DOS File Driver .....                       |          | ASCII .....                          | 565             |
| Supported File Commands and Functions ..... | 646      | Basic .....                          | 572             |
| DOS Files .....                             | 643      | Btrieve .....                        | 583             |
| Buffered access behavior .....              | 645      | Clarion .....                        | 598             |
| File Specifications .....                   | 644      | Clipper .....                        | 606             |
| QUICKSCAN .....                             | 645      | dBaseIII .....                       | 620             |
| Supported Data Types .....                  | 644      | dBaseIV .....                        | 634             |
| DOS files .....                             |          | DOS .....                            | 646             |
| Driver Strings .....                        | 645      | FoxPro .....                         | 654             |
| SEND .....                                  | 645      | MSSQL .....                          | 755             |
| Double hyphen .....                         | 223      | ODBC .....                           | 776             |

- Oracle ..... 796
  - Scalable SQL ..... 818, 830
  - TopSpeed ..... 667
  - DUP attribute ..... 78
  - DUPLICATE ..... 735
    - ASCII ..... 565
    - Basic ..... 572
    - Btrieve ..... 583
    - Clarion ..... 598
    - Clipper ..... 606
    - dBaseIII ..... 620
    - dBaseIV ..... 634
    - DOS ..... 646
    - FoxPro ..... 654
    - MSSQL ..... 755
    - ODBC ..... 776
    - Oracle ..... 796
    - Scalable SQL ..... 818, 830
    - TopSpeed ..... 667
  - duplicate field names, Clarion files ..... 600
  - Duplicated Records in BrowseBox ..... 698
  - "dynamic" INDEX ..... 88
  - Dynamic Link Library ..... 35
  - Dynamic link library ..... 234
  - Dynamically allocated memory ..... 34
- E**
- E. F. Codd ..... 77
  - early binding ..... 61
  - edit the template source code ..... 366
  - editor ..... 147
  - Editor Options ..... 147
  - Editor Tabs ..... 148
  - ellipse ..... 333
  - EMBED ..... 216, 425, **437**
  - embed ..... 358
  - embed point ..... 364, 391, 393
  - embed point availability ..... 400
  - embed point existence ..... 494
  - Embed Points ..... 361, 388
  - embed points ..... 534, 537, 539
  - Embed subsection ..... 216
  - embedded source code ..... 389
  - Embedded Source dialog ..... 216
  - embedded source point ..... 388
  - Embedded spaces ..... 226
  - Embedded SQL ..... 112, 700, 701, 775
    - Oracle ..... 795
  - Embeds button ..... 361
  - EMPTY ..... 735
    - ASCII ..... 565
  - Basic ..... 572
  - Btrieve ..... 583
  - Clarion ..... 598
  - Clipper ..... 606
  - dBaseIII ..... 620
  - dBaseIV ..... 634
  - DOS ..... 646
  - FoxPro ..... 654
  - MSSQL ..... 755
  - ODBC ..... 776
- Basic ..... 572
  - Btrieve ..... 583
  - Clarion ..... 598
  - Clipper ..... 606
  - dBaseIII ..... 620
  - dBaseIV ..... 634
  - DOS ..... 646
  - FoxPro ..... 654
  - MSSQL ..... 755
  - ODBC ..... 776
  - Oracle ..... 796
  - Scalable SQL ..... 818, 830
  - TopSpeed ..... 667
- Empty string ..... 225
  - enable/disable prompts ..... 428
  - Encapsulation ..... 60
  - ENCRYPT ..... 735
    - ASCII ..... 565
    - Basic ..... 572
    - Btrieve ..... 583
    - Clarion ..... 598
    - Clipper ..... 606
    - dBaseIII ..... 620
    - dBaseIV ..... 634
    - DOS ..... 646
    - FoxPro ..... 654
    - MSSQL ..... 755
    - ODBC ..... 776
    - Oracle ..... 796
    - Scalable SQL ..... 818, 830
    - TopSpeed ..... 667
  - Encrypted tables, logging ..... 558, 705
  - Encryption ..... 168
  - ENDOFRECORD ..... 562
    - ASCII ..... 562
    - Basic ..... 569
  - ENDOFRECORDINQUOTE ..... 569
    - Basic ..... 569
  - entry controls ..... 360
  - Enumerated ..... 257
  - EOF ..... 735
    - ASCII ..... 565
    - Basic ..... 572
    - Btrieve ..... 583
    - Clarion ..... 598
    - Clipper ..... 606
    - dBaseIII ..... 620
    - dBaseIV ..... 634
    - DOS ..... 646
    - FoxPro ..... 654
    - MSSQL ..... 755
    - ODBC ..... 776

|                                            |               |                                                          |                    |
|--------------------------------------------|---------------|----------------------------------------------------------|--------------------|
| Oracle .....                               | 796           | Example—Relation Definition Group .....                  | 189                |
| Scalable SQL .....                         | 818, 830      | Exclamation point .....                                  | 190, 191, 206      |
| TopSpeed .....                             | 667           | Exe .....                                                | 230                |
| Error 47 .....                             | 804           | Executable code section .....                            | 33                 |
| Error Handling (SQL) .....                 | 116           | ExecuteProject .....                                     | 159                |
| Error message .....                        | 242           | EXETYPE .....                                            | 287, 295           |
| ERROR Messages .....                       | 556           | EXISTS .....                                             | <b>494</b>         |
| Btrieve .....                              | 589           | expand a symbol .....                                    | 359                |
| Clipper .....                              | 613           | expansion macros .....                                   | 123, 129           |
| dBaseIII .....                             | 626           | Expansion Symbols .....                                  | 503                |
| dBaseIV .....                              | 641           | Export .....                                             | 553                |
| FoxPro .....                               | 661           | export a dictionary to text format .....                 | 153                |
| ODBC .....                                 | 779           | export an Application to text format .....               | 155                |
| Error Messages                             |               | export symbol to text .....                              | 487                |
| Cannot Load the (AS400) Driver .....       | 740           | ExportApp .....                                          | 155                |
| CWBSY__ Internal Error .....               | 740           | ExportDct .....                                          | 153                |
| Specified Driver Could not be Loaded ..... | 740           | Exported .....                                           | 234                |
| Unsupported Function .....                 | 739           | Exporting CLASSES .....                                  | 296, 302           |
| ERRORCODE .....                            | 116           | EXPORTS .....                                            | 287, 295, 297, 301 |
| TopSpeed .....                             | 668, 670, 677 | EXPR .....                                               | <b>425, 437</b>    |
| ERRORCODE 80                               |               | Expression in delete[] is obsolete .....                 | 272                |
| SQL Drivers .....                          | 693           | Expressions .....                                        | 236                |
| EVALUATE() .....                           | 412, 421      | extensible supplemental utilities .....                  | 378                |
| Evaluation of complete expressions .....   | 260           | Extension .....                                          | 227                |
| Evaluation order .....                     | 236           | Extension Template .....                                 | 546                |
| event handling loop .....                  | 530           | Extension template .....                                 | 219                |
| EVENT:Accepted .....                       | 539           | extension template .....                                 | 362, 365, 366, 385 |
| EVENT:GainFocus .....                      | 534           | Extension templates .....                                | 359                |
| EVENT:OpenWindow .....                     | 534           | extension templates .....                                | 358, 361, 537      |
| EVENT:Selected .....                       | 539           | Extern declarations .....                                | 258                |
| EVENT:Timer .....                          | 549           | EXTERNAL .....                                           | 321, 374, 735      |
| Example File                               |               | ASCII .....                                              | 565                |
| TopSpeed .....                             | 678           | Basic .....                                              | 572                |
| TPSFIX .....                               | 677           | Btrieve .....                                            | 583                |
| Example Program                            |               | Clarion .....                                            | 598                |
| running .....                              | 742           | Clipper .....                                            | 606                |
| Example—[ADDITION] .....                   | 220           | dBaseIII .....                                           | 620                |
| Example—[APPLICATION] .....                | 197           | dBaseIV .....                                            | 634                |
| Example—[COMMON] .....                     | 205           | DOS .....                                                | 646                |
| Example—[DATA] .....                       | 207           | FoxPro .....                                             | 654                |
| Example—[DICTIONARY] .....                 | 169           | MSSQL .....                                              | 755                |
| Example—[EMBED]-[END] .....                | 217           | ODBC .....                                               | 776                |
| Example—[FILES] .....                      | 210           | Oracle .....                                             | 796                |
| Example—[PROCEDURE]-[END] .....            | 203           | Scalable SQL .....                                       | 818, 830           |
| Example—[PROGRAM]-[END] .....              | 200           | TopSpeed .....                                           | 667                |
| Example—Alias Definition Group .....       | 185           | External Name field .....                                | 725                |
| Example—Blob Definition Group .....        | 178           | External Source .....                                    | 305                |
| Example—Field Definition Group .....       | 183           | EXTRACT .....                                            | 363, <b>493</b>    |
| Example—File Definition Group .....        | 171           |                                                          |                    |
| Example—Key Definition Group .....         | 173           | <b>F</b>                                                 |                    |
| Example—Memo Definition Group .....        | 177           | Failed to re-open in exclusive mode TopSpeed files ..... | 670                |
| Example—Module Definition File .....       | 288, 297      |                                                          |                    |

- FAMILY ..... **369**
- Far to near pointer conversion ..... 269
- fat fetches ..... 112
- FIELD ..... 213, 425, **438**
- Field Definition ..... 180, 207
- Field Definition Group ..... 179
- Field Properties dialog ..... 175, 181, 207
- field-independent event ..... 533
- field-independent events ..... 530
- field-specific events ..... 531, 535, 536
- FIELDDELIMITER
  - Basic ..... 570
- FILE ..... 213, 425, **438**
- File Access ..... 87
- File access ..... 209
- file control block ..... 328
- File date ..... 234
- File Definition Group ..... 170
- file driver functions ..... 328
- File Driver Symbols ..... 522
- file drivers ..... 553
- file existence ..... 494
- file I/O ..... 357, 713
- file I/O, ODBC ..... 761
- file I/O, SQL Drivers ..... 687
- file lookup code ..... 541
- File Management ..... 241
- File Manipulation ..... 351
- File Masks ..... 126
- File Not Found ..... 803
- File Relationships ..... 79
- File Schematic Symbols ..... 521
- File Specification Maximums ..... 732
- File Specifications, ASCII ..... 561
- File Types ..... 126
- FILE\_ALIGNMENT ..... 297, 301
- FILE\_TO\_RELATED\_KEY ..... 187
- FILEBUFFERS
  - ASCII ..... 563
  - Basic ..... 570
  - DOS ..... 645
- FileDrop control template ..... 543
- FILEERROR ..... 116
  - SQL Drivers ..... 556, 704
- FILEERRORCODE ..... 116
  - SQL Drivers ..... 556, 704
- FILEEXISTS ..... **494**
- Filename ..... 129, 227
- Filenames ..... 232
- Files
  - Variable file names ..... 584, 599, 668, 777
- Files and Editing ..... 224
- Files section ..... 170
- Files subsection ..... 209
- filter ..... 722, 746, 758, 821, 834
- FILTER attribute ..... 111
- filter out unwanted records ..... 111
- FILTER the result set ..... 110
- Find files ..... 239
- FINISH ..... **448**
- FIRST ..... **380, 382, 386, 391**
- First Aid ..... 165, 193
- first file ..... 743
- fix a multi-value symbol ..... 418, 420, 421
- Floating-point data ..... 318
- Flow Control ..... 236
- FLUSH ..... 735
  - ASCII ..... 565
  - Basic ..... 572
  - Btrieve ..... 583
  - Clarion ..... 598
  - Clipper ..... 606
  - dBaseIII ..... 620
  - dBaseIV ..... 634
  - DOS ..... 646
  - FoxPro ..... 654
  - MSSQL ..... 755
  - ODBC ..... 776
  - Oracle ..... 796
  - Scalable SQL ..... 818, 830
  - TopSpeed ..... 667
- fnmerge ..... 352
- fnsplit ..... 352
- font= ..... 147
- FOR ..... 454
- FORCEUPPERCASE
  - ODBC ..... 770
- Foreign Key ..... 78, 108
- FORMAT ..... 425, **439**
- Format a symbol ..... 503
- Formula ..... 203
- formula class ..... 484
- formula classes ..... 526
- Formula Symbols ..... 520
- FoxPro
  - non-english versions ..... 655
- FoxPro File Driver
  - Supported File Commands and Functions ..... 654
- FoxPro files ..... 649
  - Data Types ..... 649
  - File Specifications ..... 651
- free a multi-valued symbol ..... 417
- Free-form text ..... 217
- Frequently used constants ..... 260

|                                                 |                           |
|-------------------------------------------------|---------------------------|
| friends .....                                   | 67                        |
| FROM .....                                      | 205, 425, 429, <b>439</b> |
| Full Pathname field .....                       | 724, 726                  |
| FULLBUILD                                       |                           |
| TopSpeed .....                                  | 665                       |
| TopSpeed Keys .....                             | 683, 684                  |
| FULLNAME .....                                  | 495                       |
| Function not declared .....                     | 269                       |
| Function overloading .....                      | 60                        |
| function overloading .....                      | 70                        |
| Function prototype not declared .....           | 269                       |
| Function prototypes .....                       | 304                       |
| Function redeclared with fixed parameters ..... | 272                       |
| functions .....                                 | 327                       |
| Future Oracle Releases .....                    | 781, 800                  |

**G**

|                                                   |                         |
|---------------------------------------------------|-------------------------|
| GATHERATOPEN .....                                | 730, 751, 771, 814, 827 |
| generate source code section .....                | 473                     |
| generate the source code for an Application ..... | 158                     |
| GenerateApp .....                                 | 158                     |
| generated code .....                              | 716, 784                |
| GenerateUtilityTemplate .....                     | 160                     |
| GET .....                                         | 91, 735                 |
| ASCII .....                                       | 565                     |
| Basic .....                                       | 572                     |
| Btrieve .....                                     | 583                     |
| Clarion .....                                     | 598                     |
| Clipper .....                                     | 606                     |
| dBaseIII .....                                    | 620                     |
| dBaseIV .....                                     | 634                     |
| DOS .....                                         | 646                     |
| FoxPro .....                                      | 654                     |
| MSSQL .....                                       | 755                     |
| ODBC .....                                        | 776                     |
| Oracle .....                                      | 796                     |
| Scalable SQL .....                                | 818, 830                |
| TopSpeed .....                                    | 667                     |
| get all the related records .....                 | 532                     |
| getcurdir .....                                   | 353                     |
| getdisk .....                                     | 353                     |
| GetErrorMsg .....                                 | 162                     |
| GetErrorNum .....                                 | 162                     |
| GETINI .....                                      | 119                     |
| global area .....                                 | 373                     |
| Global data declaration section .....             | 34                      |
| global data section .....                         | 407                     |
| Global data threshold .....                       | 253, 256                |
| Global initialized data .....                     | 253                     |
| Global Map .....                                  | 304                     |
| Global prompts .....                              | 373                     |

|                                 |               |
|---------------------------------|---------------|
| Global Properties window .....  | 371           |
| global section .....            | 364           |
| Global Settings window .....    | 526           |
| Global uninitialized data ..... | 253           |
| GlobalRequest .....             | 541           |
| GlobalRequest variable .....    | 530           |
| GlobalResponse .....            | 541           |
| grid= .....                     | 141, 145      |
| GROUP .....                     | 306, 308, 313 |
| Groupware .....                 | 105           |
| GUI .....                       | 288, 298      |

**H**

|                                                          |                 |
|----------------------------------------------------------|-----------------|
| hard disk space .....                                    | 718             |
| hdbc, ODBC .....                                         | 773             |
| HEADER .....                                             | 176, 182        |
| header .....                                             | 129             |
| HEAP_COMMIT .....                                        | 297, 299        |
| HEAP_RESERVE .....                                       | 297, 300        |
| HEAPSIZE .....                                           | 287, 294        |
| HELD                                                     |                 |
| Clarion .....                                            | 595             |
| HELP .....                                               | 176, 182        |
| help file .....                                          | 487             |
| henv, ODBC .....                                         | 773             |
| HIDE .....                                               | 388, 445        |
| HIDE attribute .....                                     | 535, 537        |
| hierarchies of classes .....                             | 60              |
| HINT                                                     |                 |
| Oracle .....                                             | 792             |
| HLP ... 196, 371, 373, 374, 375, 376, 378, 379, 388, 429 |                 |
| HLP attributes .....                                     | 487             |
| HOLD .....                                               | 99, 735         |
| ASCII .....                                              | 565             |
| Basic .....                                              | 572             |
| Btrieve .....                                            | 583             |
| Clarion .....                                            | 598             |
| Clipper .....                                            | 606             |
| dBaseIII .....                                           | 620             |
| dBaseIV .....                                            | 634             |
| DOS .....                                                | 646             |
| FoxPro .....                                             | 654             |
| MSSQL .....                                              | 755             |
| ODBC .....                                               | 776             |
| Oracle .....                                             | 796             |
| Scalable SQL .....                                       | 818, 830        |
| TopSpeed .....                                           | 667             |
| HSCROLL .....                                            | <b>424, 447</b> |
| hstmt, ODBC .....                                        | 773             |
| Hyphen .....                                             | 223             |

- I**
- ICON ..... 423
  - IDENT ..... 175, 181
  - IDX, FoxPro ..... 649
  - If You're New to Clarion... 784
  - If You're New to Oracle... 785
  - IGNORECORRUPTIONS
    - Clarion ..... 595
  - IGNORESTATUS
    - Clarion ..... 596
    - Clipper ..... 605
    - dBaseIII ..... 619
    - dBaseIV ..... 633
    - FoxPro ..... 653
  - IMAGE\_BASE ..... 297, 300
  - IMPORT ..... 233
  - import ..... 553
  - import .APP from script ..... 488
  - import a dictionary from text format ..... 154
  - import an Application from text ..... 156
  - Import library ..... 234
  - Import Table Definition ..... 723
  - Import Wizard ..... 721
  - Import Wizard, Oracle ..... 786
  - Import Wizard, Scalable SQL ..... 745, 809
  - Import Wizard, SQLAnywhere ..... 824
  - ImportApp ..... 156
  - ImportDct ..... 154
  - Importing File Definitions ..... 554
  - Importing Oracle Files to a Data Dictionary ..... 721
  - IN clause (SQL) ..... 114
  - INCLUDE ..... 199
  - include ..... 327
  - include a template file ..... 488
  - Include file ..... 304
  - INCLUDE('dde.clw') ..... 150
  - incrementX= ..... 130
  - incrementY= ..... 130
  - indentation level ..... **458**
  - INDEX ..... 87, 735
    - ASCII ..... 565
    - Basic ..... 572
    - Btrieve ..... 583
    - Clarion ..... 598
    - Clipper ..... 606
    - dBaseIII ..... 620
    - dBaseIV ..... 634
    - DOS ..... 646
    - FoxPro ..... 654
    - MSSQL ..... 755
    - ODBC ..... 776
    - Oracle ..... 796
    - Scalable SQL ..... 818, 830
    - TopSpeed ..... 667
  - Index Record Number ..... 90
  - Index, SQL Drivers ..... 695
  - INFILE ..... 176, 182
  - Inheritance ..... 60
  - init\_priority ..... 277
  - INITGLOBAL ..... 299
  - INITIAL ..... 176, 182
  - Initial value ..... 176, 182
  - Initialization code ..... 264
  - Initialization order ..... 264
  - InitializeWindow ROUTINE ..... 532
  - INITINSTANCE ..... 299
  - INLINE ..... **424, 430**
  - Inline ..... 252
  - Inline function ..... 249
  - INLIST ..... 176, 182, **495**
  - INNER attribute ..... 111
  - inner join ..... 111
  - Input and Validation Statements ..... 423
  - input from the developer ..... 357
  - Input Validation Statements ..... 362
  - INRANGE ..... 176, 181
  - INS ..... 177, 183
  - insert code from a #GROUP ..... 459, 460, 461
  - insert= ..... 147
  - Installation ..... 714, 783
  - INSTANCE ..... 217, 219, 420, **496**
  - Instance ..... 216
  - INSTANCE(%symbol) ..... 414
  - INTEGER ..... 732
  - integer ..... 334
  - Integer Math ..... 344
  - Interfaces to a different language ..... 264
  - Interfacing to Pascal ..... 258
  - Interfacing to third party C ..... 258
  - Interfacing to third-party libraries ..... 255
  - Introduction ..... 713
  - Invalid Field Type Descriptor ..... 803
  - INVOKE
    - Built-in Template function ..... **497**
  - IPX host ..... 786, 789
  - isalpha ..... 345
  - isascii ..... 345
  - iscntrl ..... 345
  - isdigit ..... 346
  - isgraph ..... 346
  - islower ..... 345
  - ISNULL
    - SQL Drivers ..... 693

|                                  |                                   |                                          |                                                        |
|----------------------------------|-----------------------------------|------------------------------------------|--------------------------------------------------------|
| isprint .....                    | 346                               | Keyword 'overload' is not required ..... | 272                                                    |
| ispunct .....                    | 346                               | Keyword Parameters .....                 | 224                                                    |
| isspace .....                    | 346                               | Keywords .....                           | 166, 194, 223                                          |
| isupper .....                    | 345                               |                                          |                                                        |
| isxdigit .....                   | 346                               | <b>L</b>                                 |                                                        |
| item exists in list .....        | 495                               | LABEL .....                              | <b>389</b>                                             |
| ITEMS .....                      | <b>498</b>                        | labs .....                               | 344                                                    |
| iteratively generate code .....  | 454                               | Language-dependent options .....         | 265                                                    |
|                                  |                                   | LANs .....                               | 93                                                     |
| <b>J</b>                         |                                   | LAST .....                               | 171, <b>380</b> , <b>382</b> , <b>386</b> , <b>391</b> |
| JOIN                             |                                   | late binding .....                       | 61                                                     |
| ODBC .....                       | 762                               | left justified .....                     | 340                                                    |
| JOIN, dynamic (runtime) .....    | 710                               | left outer join .....                    | 110                                                    |
| Join file .....                  | 80                                | LegacyAction= .....                      | 136                                                    |
| JOIN operations .....            | 110                               | length .....                             | 341                                                    |
| JOINTYPE                         |                                   | Lib .....                                | 230                                                    |
| ODBC .....                       | 771                               | LIBRARY .....                            | 287, 289, 297, 299                                     |
| journalling .....                | 730                               | LIKE .....                               | 309                                                    |
| Jumps .....                      | 261                               | line continuation character .....        | 503, 525                                               |
|                                  |                                   | Line number .....                        | 263                                                    |
| <b>K</b>                         |                                   | LINENUMBERS .....                        | 297, 300                                               |
| KEY .....                        | 87, 213, 425, <b>440</b> , 735    | lines= .....                             | 128                                                    |
| ASCII .....                      | 565                               | Link Commands .....                      | 232                                                    |
| Basic .....                      | 572                               | Link list .....                          | 228, 232, 233, 234, 274                                |
| Btrieve .....                    | 583                               | Link Options .....                       | 230                                                    |
| Clarion .....                    | 598                               | Linkage naming .....                     | 258                                                    |
| Clipper .....                    | 606                               | Linker .....                             | 223                                                    |
| dBaselII .....                   | 620                               | Linker options .....                     | 274                                                    |
| dBaselIV .....                   | 634                               | Linking .....                            | 230                                                    |
| DOS .....                        | 646                               | LINKNAME .....                           | 498                                                    |
| FoxPro .....                     | 654                               | list of choices for input .....          | 362                                                    |
| MSSQL .....                      | 755                               | list of data fields .....                | 437, 438, 441                                          |
| ODBC .....                       | 776                               | list of files .....                      | 438                                                    |
| Oracle .....                     | 796                               | list of KEY fields .....                 | 435                                                    |
| Scalable SQL .....               | 818, 830                          | list of keycodes .....                   | 440                                                    |
| SQL Drivers .....                | 692, 695                          | list of keys .....                       | 440                                                    |
| TopSpeed .....                   | 667                               | list of symbol values .....              | 439                                                    |
| Key .....                        | 210                               | list of window fields .....              | 436                                                    |
| Key component .....              | 213                               | listbox formatter .....                  | 439                                                    |
| KEY Definition .....             | 172                               | Load an Application .....                | 157                                                    |
| Key Definition Group .....       | 171                               | LoadApplication .....                    | 157                                                    |
| Key Names .....                  | 725                               | Local Area Networks .....                | 93                                                     |
| KEY optimization .....           | 695                               | local host .....                         | 789                                                    |
| Key Properties dialog .....      | 791                               | Local Procedures .....                   | 38                                                     |
| Key Rebuild TopSpeed Files ..... | 683, 684                          | local variable declarations .....        | 529                                                    |
| Keyboard entry .....             | 237                               | Local variable never used .....          | 272                                                    |
| KEYCODE .....                    | 425, <b>440</b>                   | local variables .....                    | 526, 544                                               |
| Keys, full build .....           | 665                               | LocalMap= .....                          | 135                                                    |
| Keys, SQL Drivers .....          | 688                               | LocalResponse .....                      | 541                                                    |
| Keyword List .....               | 171, 173, 175, 178, 180, 185, 207 | Locate                                   |                                                        |
|                                  |                                   | TopSpeed .....                           | 679                                                    |

LOCK ..... 101, 735  
   ASCII ..... 565  
   Basic ..... 572  
   Btrieve ..... 583  
   Clarion ..... 598  
   Clipper ..... 606  
   dBaseIII ..... 620  
   dBaseIV ..... 634  
   DOS ..... 646  
   FoxPro ..... 654  
   MSSQL ..... 755  
   ODBC ..... 776  
   Oracle ..... 796  
   Scalable SQL ..... 818, 830  
   TopSpeed ..... 667  
 logarithm ..... 334  
 LOGFILE  
   SQL Drivers ..... 705  
 Logging In ..... 728  
 Logic Control ..... 451  
 logical\_round ..... 277  
 Login Dialog, SQL Drivers ..... 692  
 LogInWindow ..... 742  
 logon, MSSQL ..... 731, 753  
 logon, ODBC ..... 774  
 logon, Scalable SQL ..... 815  
 logon, SQLAnywhere ..... 828  
 LOGONSCREEN  
   MSSQL ..... 751  
   Scalable SQL ..... 814  
   SQLAnywhere ..... 827  
 LOGOUT ..... 730, 735  
   ASCII ..... 565  
   Basic ..... 572  
   Btrieve ..... 583  
   Clarion ..... 598, 600  
   Clipper ..... 606  
   dBaseIII ..... 620  
   dBaseIV ..... 634  
   DOS ..... 646  
   FoxPro ..... 654  
   MSSQL ..... 755  
   ODBC ..... 776  
   Oracle ..... 796  
   Scalable SQL ..... 818, 830  
   SQL Drivers ..... 698  
   TopSpeed ..... 667  
 LONG ..... 204, 212, 306, 732  
 LOOP ..... 88, 89  
   loop structure ..... 451  
 Lotus Notes and ODBC ..... 762  
 lower case ..... 341

**M**

M ..... 245  
 Machine registers ..... 249  
 Macro names ..... 225  
 Macros ..... 223, 225, 227  
 Main ..... 197, 228  
   main program module ..... 373  
   Main source file ..... 228  
   maincode ..... 276  
 MAINTAINHEADERTIME  
   Clarion ..... 596  
 Make ..... 223  
 Make File Types ..... 129  
 Many-to-Many ..... 79  
 Many-to-One ..... 79  
 MAP ..... 388  
 Map ..... 327  
 MAP file ..... 275  
 Map file ..... 274  
 MAP structure ..... 34  
 Mass Changes ..... 165, 193  
 matched set of controls ..... 382  
 maximized= ..... 120, 147  
 MDX, dBaseIV ..... 629  
 MEMBER ..... 35  
 MEMBER data declaration section ..... 37  
 MEMBER MAPs ..... 36  
 MEMBER module ..... 35  
 MEMO ..... 735  
   ASCII ..... 565  
   Basic ..... 572  
   Btrieve ..... 583  
   Clarion ..... 598  
   Clipper ..... 606  
   dBaseIII ..... 620  
   dBaseIV ..... 634  
   DOS ..... 646  
   FoxPro ..... 654  
   MSSQL ..... 755  
   ODBC ..... 776  
   Oracle ..... 796  
   Scalable SQL ..... 818, 830  
   TopSpeed ..... 667  
 MEMO Definition ..... 174  
 Memo Definition Group ..... 173  
 Memory model ..... 231, 249  
 Memory variable ..... 205  
 MEMOs versus STRINGS, TopSpeed files ..... 668, 672  
 MESSAGE ..... 176, 182  
 MessageLines= ..... 140  
 messages ..... 266

|                                          |               |                                                       |                              |
|------------------------------------------|---------------|-------------------------------------------------------|------------------------------|
| method .....                             | 59            | LOGONSCREEN .....                                     | 751                          |
| Method table .....                       | 255           | SAVSTOREDPROC .....                                   | 751                          |
| methods .....                            | 62            | Supported File Commands and Functions .....           | 755                          |
| methods (PROCEDURES and FUNCTIONS) ..... | 62            | TRUSTEDCONNECTION .....                               | 752                          |
| Microsoft Access and ODBC .....          | 778           | MSSQL logon .....                                     | 731, 753                     |
| Microsoft C calling convention .....     | 248           | MULTI .....                                           | 212, 381, 385, 409, 423, 429 |
| Mixing languages .....                   | 304           | Multi Language Programming .....                      | 303                          |
| Mixing the use of HOLD and LOCK .....    | 103           | Multi-developer environments .....                    | 204                          |
| mkdir .....                              | 352           | Multi-language Programming .....                      | 250                          |
| MODIFIED .....                           | 168, 205      | Multi-tasking .....                                   | 93                           |
| Modula-2 .....                           | 303, 327, 329 | Multi-threading .....                                 | 93                           |
| Data Type Equivalents .....              | 310           | Multi-user environments .....                         | 93                           |
| Interface considerations .....           | 324           | multi-valued symbol instances .....                   | 498                          |
| MODULE .....                             | 34            | multi-valued symbols .....                            | 505                          |
| module area .....                        | 374           | Multimedia .....                                      | 328                          |
| MODULE attribute .....                   | 62            | Multiple Inheritance .....                            | 71                           |
| Module data .....                        | 200           | multiple servers .....                                | 727                          |
| Module definition file .....             | 235           | Multiple Tables in a single TopSpeed file .....       | 674                          |
| module definition file .....             | 287           | MultiUser= .....                                      | 134                          |
| 32-bit .....                             | 297           | <b>N</b>                                              |                              |
| CODE .....                               | 289           | NAME 199, 201, 219, 287, 288, 297, 298, 314, 320, 735 |                              |
| CODE statement .....                     | 287           | ASCII .....                                           | 565                          |
| DATA .....                               | 291           | Basic .....                                           | 572                          |
| DATA statement .....                     | 287           | Btrieve .....                                         | 583                          |
| example .....                            | 288, 297      | Clarion .....                                         | 598                          |
| EXETYPE .....                            | 295           | Clipper .....                                         | 606                          |
| EXETYPE statement .....                  | 287           | dBaseIII .....                                        | 620                          |
| EXPORTS .....                            | 295, 301      | dBaseIV .....                                         | 634                          |
| EXPORTS statement .....                  | 287           | DOS .....                                             | 646                          |
| HEAPSIZE .....                           | 294           | FoxPro .....                                          | 654                          |
| HEAPSIZE statement .....                 | 287           | MSSQL .....                                           | 755                          |
| LIBRARY .....                            | 289, 299      | ODBC .....                                            | 776                          |
| LIBRARY statement .....                  | 287           | Oracle .....                                          | 796                          |
| NAME .....                               | 288, 298      | Scalable SQL .....                                    | 818, 830                     |
| NAME statement .....                     | 287           | TopSpeed .....                                        | 667                          |
| SEGMENTS .....                           | 293           | NAME attribute .....                                  | 724, 725                     |
| SEGMENTS statement .....                 | 287           | name clashes .....                                    | 156                          |
| STACKSIZE .....                          | 294           | Name mangling .....                                   | 258                          |
| STACKSIZE statement .....                | 287           | NameClash= .....                                      | 135                          |
| Module section .....                     | 200           | "named" references .....                              | 65                           |
| MODULE structure .....                   | 305           | Naming Conventions .....                              | 314, 319                     |
| ModuleProcs= .....                       | 135           | Naming object files .....                             | 264                          |
| MODULEs within MEMBER MAPs .....         | 37            | natural logarithm .....                               | 334                          |
| modulus division operator (%) .....      | 412, 421      | NDX                                                   |                              |
| modulus operator .....                   | 550           | dBaseIV .....                                         | 629                          |
| month .....                              | 334           | Near calls .....                                      | 248                          |
| mouse .....                              | 334           | Near to far pointer conversion .....                  | 270                          |
| Movable Thumbs, SQL Drivers .....        | 697           | Nested comments .....                                 | 266                          |
| Move .....                               | 242           | Nested Dependent Prompts .....                        | 214                          |
| MSSQL                                    |               | NESTING                                               |                              |
| Driver Properties .....                  | 731, 753      |                                                       |                              |
| Driver Strings .....                     | 751           |                                                       |                              |

- ODBC ..... 771
  - NESTING views and ODBC ..... 771
  - network traffic ..... 108
  - NEW ..... 64
  - New Dialog ..... 127
  - newtype= ..... 133
  - NEXT ..... 88, 735
    - ASCII ..... 565
    - Basic ..... 572
    - Btrieve ..... 583
    - Clarion ..... 598
    - Clipper ..... 606
    - dBaseIII ..... 620
    - dBaseIV ..... 634
    - DOS ..... 646
    - FoxPro ..... 654
    - MSSQL ..... 755
    - ODBC ..... 776
    - Oracle ..... 796
    - Scalable SQL ..... 818, 830
    - SQL Drivers ..... 692
    - TopSpeed ..... 667
  - NEXT statement ..... 113
  - No expression in return statement ..... 269
  - No return value in function ..... 269
  - NOCASE ..... 735
    - ASCII ..... 565
    - Basic ..... 572
    - Btrieve ..... 583
    - Clarion ..... 598
    - Clipper ..... 606
    - dBaseIII ..... 620
    - dBaseIV ..... 634
    - DOS ..... 646
    - FoxPro ..... 654
    - MSSQL ..... 755
    - ODBC ..... 776
    - Oracle ..... 796
    - Scalable SQL ..... 818, 830
    - TopSpeed ..... 667
  - NOINIDENT ..... **389, 459, 461**
  - NOLINK ..... 188
  - NOMEMO ..... 735
    - ASCII ..... 565
    - Basic ..... 572
    - Btrieve ..... 583
    - Clarion ..... 598
    - Clipper ..... 606
    - dBaseIII ..... 620
    - dBaseIV ..... 634
    - DOS ..... 646
    - FoxPro ..... 654
  - MSSQL ..... 755
  - ODBC ..... 776
  - Oracle ..... 796
  - Scalable SQL ..... 818, 830
  - SQL Drivers ..... 692
  - TopSpeed ..... 667
  - Non-local jumps ..... 252
  - NONBLANK ..... 175, 181
  - NONZERO ..... 175, 181
  - NOPOPULATE ..... 170, 172, 174, 180, 184, 199
  - NORESULTCALL
    - Oracle ..... 795
    - SQL Drivers ..... 702
  - Not ..... 238
  - NOT NULL attribute (SQL) ..... 113
  - NOWHERE
    - ODBC ..... 769
  - NT ..... 303
  - NULL ..... 115, 735
    - ASCII ..... 565
    - Basic ..... 572
    - Btrieve ..... 583
    - Clarion ..... 598
    - Clipper ..... 606
    - dBaseIII ..... 620
    - dBaseIV ..... 634
    - DOS ..... 646
    - FoxPro ..... 654
    - MSSQL ..... 755
    - ODBC ..... 776
    - Oracle ..... 796
    - Scalable SQL ..... 818, 830
    - SQL Drivers ..... 693
    - TopSpeed ..... 667
  - NULL data ..... 115
  - Null Dates & Times with ODBC ..... 772
  - NULL pointers ..... 257
  - NULL terminated strings ..... 308
  - Nullify the Foreign Key ..... 85
  - NUMERIC ..... 732
- ## O
- Object Linking and Embedding ..... 105
  - Objects ..... 59
  - ODBC ..... 107, 762
    - BINDCONSTANTS ..... 770
    - CLIPSTRINGS ..... 770
    - Data Sources ..... 763, 764
    - Driver Strings ..... 770
    - FORCEUPPERCASE ..... 770
    - JOINTYPE ..... 771
    - nested views ..... 771

|                                             |          |                                         |                         |
|---------------------------------------------|----------|-----------------------------------------|-------------------------|
| NESTING .....                               | 771      | Clarion .....                           | 598                     |
| NOWHERE .....                               | 769      | Clipper .....                           | 606                     |
| null Dates and Times .....                  | 772      | dBaseIII .....                          | 620                     |
| ODBCCALL .....                              | 771      | dBaseIV .....                           | 634                     |
| Pros and Cons .....                         | 762      | DOS .....                               | 646                     |
| READONLY .....                              | 768      | FoxPro .....                            | 654                     |
| Supported File Commands and Functions ..... | 776      | MSSQL .....                             | 755                     |
| USEINNERJOIN .....                          | 772      | ODBC .....                              | 776                     |
| VERIFYVIASELECT .....                       | 772      | Oracle .....                            | 796                     |
| views and joins .....                       | 771      | Scalable SQL .....                      | 818, 830                |
| WATCH .....                                 | 769      | TopSpeed .....                          | 667                     |
| WHERE .....                                 | 769      | Open Database Connectivity .....        | 762                     |
| ZEROISNULL .....                            | 772      | Open Dialog .....                       | 127                     |
| ODBC Administrator .....                    | 764      | Open File dialog .....                  | 441                     |
| ODBC Administrator Logs .....               | 769      | open source file .....                  | 464                     |
| ODBC driver .....                           |          | OPENDIALOG .....                        | 425, 441                |
| Microsoft Access .....                      | 778      | Operators .....                         | 237                     |
| ODBC Driver Kit v2 .....                    | 778      | OPT .....                               | 735                     |
| ODBC Driver Manager .....                   | 762      | ASCII .....                             | 565                     |
| ODBC Files .....                            |          | Basic .....                             | 572                     |
| Supported Data Types .....                  | 765      | Btrieve .....                           | 583                     |
| ODBC logon .....                            | 774      | Clarion .....                           | 598                     |
| ODBC.INI .....                              | 764      | Clipper .....                           | 606                     |
| ODBCCALL .....                              |          | dBaseIII .....                          | 620                     |
| ODBC .....                                  | 771      | dBaseIV .....                           | 634                     |
| OEM .....                                   | 554, 735 | DOS .....                               | 646                     |
| ASCII .....                                 | 565      | FoxPro .....                            | 654                     |
| Basic .....                                 | 572      | MSSQL .....                             | 755                     |
| Btrieve .....                               | 583      | ODBC .....                              | 776                     |
| Clarion .....                               | 598      | Oracle .....                            | 796                     |
| Clipper .....                               | 606      | Scalable SQL .....                      | 818, 830                |
| dBaseIII .....                              | 620      | TopSpeed .....                          | 667                     |
| dBaseIV .....                               | 634      | OPTFIELD .....                          | 425, 441                |
| DOS .....                                   | 646      | Optimization .....                      | 259                     |
| FoxPro .....                                | 654      | optimization, SQL Drivers .....         | 695                     |
| MSSQL .....                                 | 755      | optimize by cpu .....                   | 261                     |
| ODBC .....                                  | 776      | optimize performance .....              | 742                     |
| Oracle .....                                | 796      | Optimized entry .....                   | 252                     |
| Scalable SQL .....                          | 818, 830 | Optimizing for space .....              | 260                     |
| TopSpeed .....                              | 667      | Optimizing for speed .....              | 260                     |
| OLE .....                                   | 105      | OPTION .....                            | 425, 442                |
| OLTP .....                                  | 105      | Options .....                           | 170, 172, 173, 179, 184 |
| OMIT .....                                  | 276      | Or .....                                | 237                     |
| ON DELETE clause (SQL) .....                | 113      | Oracle .....                            |                         |
| On Line Transaction Processing (OLTP) ..... | 105      | auto numbered keys .....                | 790                     |
| One-to-Many .....                           | 79       | sequence numbers .....                  | 790                     |
| One-to-One .....                            | 79       | Supported Commands and Attributes ..... | 796                     |
| ONE:MANY .....                              | 186      | Supported Data Types .....              | 798                     |
| OPEN .....                                  | 93, 735  | Unique Key Values .....                 | 790                     |
| ASCII .....                                 | 565      | Oracle BatchUpdate .....                | 806                     |
| Basic .....                                 | 572      | Oracle connection .....                 | 786                     |
| Btrieve .....                               | 583      | Oracle CONSTRAINTS .....                | 786                     |

- Oracle Data Types ..... 798
- Oracle Driver Strings ..... 792
- Oracle error -921 ..... 806
- Oracle hints ..... 792
- Oracle indexes ..... 786
- Oracle linking fields ..... 793
- Oracle login ..... 786
- Oracle Login Dialog ..... 789
- Oracle LogInWindow ..... 805
- Oracle not available ..... 803
- Oracle Personal ..... 793
- Oracle Sequences ..... 790
- Oracle teaching example ..... 805
- Oracle version ..... 782
- Oracle versions ..... 781, 800
- ORDER ..... 172, 174, 180
- ORDER attribute ..... 111
- ORDER BY clause (SQL) ..... 111
- ORDER BY, SQL Drivers ..... 692
- Order files= ..... 131
- ORDER the result set ..... 110
- Organization ..... 166, 194
- Other languages ..... 303
- Other vendors ..... 249
- Out of range ..... 257
- outer join ..... 110
- OVER
  - Oracle ..... 796
- Overflow in constant expression ..... 272
- Overlay model programs ..... 244
- Overriding Inherited Methods ..... 69
- OVR ..... 177, 183
- OWNER ..... 735
  - ASCII ..... 565
  - Basic ..... 572
  - Btrieve ..... 583
  - Clarion ..... 598
  - Clipper ..... 606
  - dBaseIII ..... 620
  - dBaseIV ..... 634
  - DOS ..... 646
  - FoxPro ..... 654
  - MSSQL ..... 755
  - ODBC ..... 776
  - Oracle ..... 796
  - Scalable SQL ..... 818, 830
  - TopSpeed ..... 667
- P**
- P ..... 245
- PACK ..... 735
  - ASCII ..... 565
  - Basic ..... 572
  - Btrieve ..... 583
  - Clarion ..... 598
  - Clipper ..... 606
  - dBaseIII ..... 620
  - dBaseIV ..... 634
  - DOS ..... 646
  - FoxPro ..... 654
  - MSSQL ..... 755
  - ODBC ..... 776
  - Oracle ..... 796
  - Scalable SQL ..... 818, 830
  - TopSpeed ..... 667
  - PACKED DECIMAL ..... 732
  - Packed fields ..... 255
  - Packed segments ..... 275
  - page of prompts ..... 429
  - Parameter list inconsistent with previous call ..... 271
  - Parameter never used in function ..... 270
  - Parameter passing ..... 308, 314
  - Parameter size ..... 250
  - Parameters ..... 224
  - parameters passed ..... 459, 460, 461, 492, 497
  - parameters passed into the procedure ..... 526
  - PARENT ..... 220
  - Parent-Child ..... 79
  - Pascal ..... 303
    - Data Type Equivalents ..... 312
    - Interface considerations ..... 324
  - Pascal RECORD ..... 313
  - Pass function parameters ..... 249
  - Passed by address ..... 317
  - passed by address ..... 377
  - Passed by value ..... 317
  - passed by value ..... 376
  - Passing parameters ..... 317
  - PASSWORD ..... 168, 177, 183
  - Password ..... 726, 790
    - SQL Drivers ..... 692
  - Password, ODBC ..... 767
  - Password, SQL Drivers ..... 691
  - Password, SQLAnywhere ..... 824
  - PATH
    - Oracle ..... 782
  - Path ..... 196
  - Paths ..... 239
  - pbposh= ..... 141, 145
  - pbposw= ..... 141, 145
  - pbposx= ..... 141, 145
  - pbposy= ..... 141, 145
  - PDECIMAL ..... 306, 732

|                                               |                                   |                                                  |                         |
|-----------------------------------------------|-----------------------------------|--------------------------------------------------|-------------------------|
| percent (%) .....                             | 503                               | Scalable SQL .....                               | 818, 830                |
| Percent sign .....                            | 212, 213                          | TopSpeed .....                                   | 667                     |
| percent sign (%) .....                        | 360, 364                          | POSITION Function .....                          | 737                     |
| Percent signs .....                           | 223                               | positioning of the control .....                 | 383                     |
| Performance .....                             | 554                               | Possible use of variable before assignment ..... | 270                     |
| Performance Considerations, SQL Drivers ..... | 695                               | pound (#) .....                                  | 503                     |
| PERSONAL                                      |                                   | Pound sign .....                                 | 223                     |
| Oracle .....                                  | 793                               | pound symbol (#) .....                           | 364                     |
| PERSONAL Oracle .....                         | 793                               | PRE .....                                        | 735                     |
| Physical Record Number .....                  | 90                                | ASCII .....                                      | 565                     |
| Pick Dialog .....                             | 127                               | Basic .....                                      | 572                     |
| pick-lists .....                              | 362                               | Btrieve .....                                    | 583                     |
| PICTURE .....                                 | 177, 183, <b>425</b> , <b>442</b> | Clarion .....                                    | 598                     |
| picture .....                                 | 503                               | Clipper .....                                    | 606                     |
| Picture token .....                           | 177, 183                          | dBaseIII .....                                   | 620                     |
| PL/SQL .....                                  | 795, 806                          | dBaseIV .....                                    | 634                     |
| plus sign (+) in PROP:SQLFilter .....         | 114                               | DOS .....                                        | 646                     |
| PNM=, TopSpeed files .....                    | 666                               | FoxPro .....                                     | 654                     |
| POINTER .....                                 | 735                               | MSSQL .....                                      | 755                     |
| ASCII .....                                   | 565                               | ODBC .....                                       | 776                     |
| Basic .....                                   | 572                               | Oracle .....                                     | 796                     |
| Btrieve .....                                 | 583                               | Scalable SQL .....                               | 818, 830                |
| Clarion .....                                 | 598                               | TopSpeed .....                                   | 667                     |
| Clipper .....                                 | 606                               | Pre-defined template variables .....             | 357                     |
| dBaseIII .....                                | 620                               | pre-processes the templates .....                | 366                     |
| dBaseIV .....                                 | 634                               | pre-processing template code .....               | 360                     |
| DOS .....                                     | 646                               | Prefix public names .....                        | 258                     |
| FoxPro .....                                  | 654                               | PREPARE .....                                    | <b>389</b>              |
| MSSQL .....                                   | 755                               | Preprocessor output .....                        | 266                     |
| ODBC .....                                    | 776                               | PRESERVE .....                                   | <b>376</b> , <b>391</b> |
| Oracle .....                                  | 796                               | Preserved registers .....                        | 250                     |
| Scalable SQL .....                            | 818, 830                          | PRESS .....                                      | 341                     |
| TopSpeed .....                                | 667                               | PREVIOUS .....                                   | 88, 735                 |
| Pointer conversion .....                      | 268                               | ASCII .....                                      | 565                     |
| Pointers .....                                | 314, 317                          | Basic .....                                      | 572                     |
| polymorphic .....                             | 357, 358                          | Btrieve .....                                    | 583                     |
| Polymorphism .....                            | 60                                | Clarion .....                                    | 598                     |
| Pop .....                                     | 337, 341                          | Clipper .....                                    | 606                     |
| PopulateMain= .....                           | 134                               | dBaseIII .....                                   | 620                     |
| POSITION .....                                | 735                               | dBaseIV .....                                    | 634                     |
| ASCII .....                                   | 565                               | DOS .....                                        | 646                     |
| Basic .....                                   | 572                               | FoxPro .....                                     | 654                     |
| Btrieve .....                                 | 583                               | MSSQL .....                                      | 755                     |
| Clarion .....                                 | 598                               | ODBC .....                                       | 776                     |
| Clipper .....                                 | 606                               | Oracle .....                                     | 796                     |
| dBaseIII .....                                | 620                               | Scalable SQL .....                               | 818, 830                |
| dBaseIV .....                                 | 634                               | SQL Drivers .....                                | 692                     |
| DOS .....                                     | 646                               | TopSpeed .....                                   | 667                     |
| FoxPro .....                                  | 654                               | PREVIOUS Statement .....                         | 736                     |
| MSSQL .....                                   | 755                               | PRIMARY .....                                    | 375, 379, 381, 385, 735 |
| ODBC .....                                    | 776                               | ASCII .....                                      | 565                     |
| Oracle .....                                  | 796                               | Basic .....                                      | 572                     |

- Btrieve ..... 583
- Clarion ..... 598
- Clipper ..... 606
- dBaseIII ..... 620
- dBaseIV ..... 634
- DOS ..... 646
- FoxPro ..... 654
- MSSQL ..... 755
- ODBC ..... 776
- Oracle ..... 796
- Scalable SQL ..... 818, 830
- TopSpeed ..... 667
- Primary Key ..... 78, 108
- print a source file ..... 472
- Print Specifications ..... 129
- PrintCustomer ..... 744
- printing ..... 129
- PRIORITY ..... **379, 381, 386, 391**
- Priority ..... 264
- PRIVATE ..... 66
  - Template attribute ..... **369**
- private variables ..... 36
- PROCEDURE ..... 33, 197, 213, 385, 425, **442**
- PROCEDURE attribute ..... 548
- Procedure call ..... 217
- PROCEDURE MAPs ..... 38
- procedure name clashes ..... 156
- Procedure overloading ..... 60
- procedure overloading ..... 70
- Procedure Properties ..... 526
- Procedure Properties dialog ..... 220, 360, 361, 366, 367
- Procedure Properties window ..... 362
- procedure prototype ..... 490
- PROCEDURE Prototypes ..... 34
- Procedure Setup ..... 530
- Procedure subsection ..... 201
- Procedure Template ..... 525
- procedure template ..... 358, 362, 364, 375
- Procedure templates ..... 359
- Procedure tracing ..... 263
- ProcedureWizard= ..... 135
- process multiple records, SQL Drivers ..... 692
- PROCPROP ..... 220
- profile ..... 277
- PROGRAM ..... 34
  - program documentation ..... 378
- PROGRAM MAP ..... 34
- Program section ..... 199
- PROGRAM statement ..... 199
- Program Structure ..... 33
- Programs to execute ..... 243
- PROJECT
  - ODBC ..... 762
  - Project file ..... 491
  - Project filename ..... 229
  - PROJECT operations ..... 110
  - project section ..... 198
  - Project Syatem Options ..... 133
  - Project System ..... 223
  - Project system ..... 165, 193
  - PROMPT ..... 176, 182, 357
  - PROMPT entry types ..... 357
  - Prompt Entry types ..... 362
  - prompt group box ..... 445
  - prompt pages ..... 362
  - PROMPTAT ..... 423
  - Prompts ..... 220
  - Prompts subsection ..... 212
  - PROP:Alias, SQL Drivers ..... 707
  - PROP:AlwaysRebind, ODBC ..... 773
  - PROP:AppendBuffer ..... 736
  - PROP:ConnectionString, SQL Drivers ..... 707
  - PROP:Details, SQL Drivers ..... 707
  - PROP:Disconnect, SQL Drivers ..... 707
  - PROP:hdbc, ODBC ..... 773
  - PROP:henv, ODBC ..... 773
  - PROP:Hint, Oracle ..... 794
  - PROP:hstmt, ODBC ..... 773
  - PROP:Inner, SQL Drivers ..... 708
  - PROP:Log, SQL Drivers ..... 708
  - PROP:Logfile, SQL Drivers ..... 708
  - PROP:LoginTimeout, ODBC ..... 774
  - PROP:LoginTimeout, ODBC ..... 774
  - PROP:LoginTimeout, SQL Drivers ..... 708
  - PROP:LogonScreen, MSSQL ..... 731, 753
  - PROP:LogonScreen, ODBC ..... 774
  - PROP:LogonScreen, Scalable SQL ..... 815
  - PROP:LogonScreen, SQLAnywhere ..... 828
  - PROP:OrderAllTables, SQL Drivers ..... 708
  - PROP:PositionBlock, Btrieve ..... 582
  - PROP:Profile, SQL Drivers ..... 709
  - PROP:QuoteString, ODBC ..... 774
  - PROP:SQL ..... 113
  - PROP:SQL, SQL Drivers ..... 700
  - PROP:SQLDriver ..... 559
  - PROP:SQLFilter ..... 114
  - PROP:SQLFilter, SQL Drivers ..... 709
  - PROP:SQLJoinExpression, SQL Drivers ..... 710
  - PROP:SQLOrder, SQL Drivers ..... 711
  - Proper procedure ..... 251
  - properties ..... 62
  - properties (data members) ..... 62
  - properties and behaviors ..... 59
  - properties and methods ..... 60

|                                 |                 |                                                |               |
|---------------------------------|-----------------|------------------------------------------------|---------------|
| properties of a class .....     | 60              | rand .....                                     | 347           |
| PROTECTED .....                 | 66              | random .....                                   | 335           |
| protocol                        |                 | Random access .....                            | 87            |
| local, IPX, TCP/IP .....        | 789             | Random File Access .....                       | 91            |
| PROTOTYPE .....                 | 201             | Range check .....                              | 257           |
| Prototype .....                 | 34              | RAW .....                                      | 307, 315      |
| Prototypes .....                | 304             | Oracle .....                                   | 798           |
| prototypes .....                | 327             | re-generated templates from the registry ..... | 358           |
| Prototyping API Functions ..... | 804             | Re-register When Changed .....                 | 366           |
| Prototyping Functions .....     | 314             | READ .....                                     | 464, 465      |
| PSTRING .....                   | 306             | read one line of a source file .....           | 466           |
| PString .....                   | 338             | Read Only .....                                | 93            |
| Public .....                    | 66              | read-ahead buffer .....                        | 111           |
| Public data .....               | 321             | read-only mode .....                           | 464           |
| Public names .....              | 235, 258        | Read/Write .....                               | 93            |
| Public symbols .....            | 275             | readme .....                                   | 783           |
| Push .....                      | 338, 341        | READONLY .....                                 | 177, 183, 204 |
| PUT .....                       | 735             | ODBC .....                                     | 768           |
| ASCII .....                     | 565             | REAL .....                                     | 213, 306, 732 |
| Basic .....                     | 572             | RECLAIM .....                                  | 735           |
| Btrieve .....                   | 583             | ASCII .....                                    | 565           |
| Clarion .....                   | 598             | Basic .....                                    | 572           |
| Clipper .....                   | 606             | Btrieve .....                                  | 583           |
| dBaseIII .....                  | 620             | Clarion .....                                  | 598           |
| dBaseIV .....                   | 634             | Clipper .....                                  | 606           |
| DOS .....                       | 646             | dBaseIII .....                                 | 620           |
| FoxPro .....                    | 654             | dBaseIV .....                                  | 634           |
| MSSQL .....                     | 755             | DOS .....                                      | 646           |
| ODBC .....                      | 776             | FoxPro .....                                   | 654           |
| Oracle .....                    | 796             | MSSQL .....                                    | 755           |
| Scalable SQL .....              | 818, 830        | ODBC .....                                     | 776           |
| TopSpeed .....                  | 667             | Oracle .....                                   | 796           |
| PUTINI .....                    | 119             | Scalable SQL .....                             | 818, 830      |
|                                 |                 | TopSpeed .....                                 | 667           |
| <b>Q</b>                        |                 | RECORD                                         |               |
| QUICK .....                     | <b>375</b>      | ASCII .....                                    | 565           |
| Quick-Scan Records, ASCII ..... | 564             | Basic .....                                    | 572           |
| QUICKSCAN                       |                 | Btrieve .....                                  | 583           |
| ASCII .....                     | 564             | Clarion .....                                  | 598           |
| Basic .....                     | 570             | Clipper .....                                  | 606           |
| DOS .....                       | 645             | dBaseIII .....                                 | 620           |
| quickstart= .....               | 120             | dBaseIV .....                                  | 634           |
| QUOTE .....                     | <b>499</b>      | DOS .....                                      | 646           |
| Basic .....                     | 571             | FoxPro .....                                   | 654           |
|                                 |                 | Oracle .....                                   | 796           |
| <b>R</b>                        |                 | TopSpeed .....                                 | 667           |
| RADIO .....                     | 425, <b>443</b> | Record Count, SQL Drivers .....                | 696           |
| radio button .....              | 443             | Record Definition .....                        | 179           |
| radio buttons .....             | 360, 362        | “record lock” .....                            | 99            |
| RAM .....                       | 718             | Record pointer .....                           | 92            |
|                                 |                 | RECORDS .....                                  | 735           |
|                                 |                 | ASCII .....                                    | 565           |

- Basic ..... 572
- Btrieve ..... 583
- Clarion ..... 598
- Clipper ..... 606
- dBaseIII ..... 620
- dBaseIV ..... 634
- DOS ..... 646
- FoxPro ..... 654
- MSSQL ..... 755
- ODBC ..... 776
- Oracle ..... 796
- Scalable SQL ..... 818, 830
- TopSpeed ..... 667
- RECORDS Function ..... 737
- RECOVER
  - Clarion ..... 596
  - Clipper ..... 604
  - dBaseIII ..... 618
  - dBaseIV ..... 632
  - FoxPro ..... 652
- Recreate deleted templates= ..... 140
- Recursion ..... 34
- recursive #GROUPs ..... 390
- recursive classes ..... 72
- Redirection ..... 226, 242
- Redirection Analysis ..... 241
- Redirection file ..... 196
- Reducing network traffic, SQL Drivers ..... 695
- reference to a specific QUEUE, GROUP, or C ..... 65
- reference variable ..... 64
- reference variables ..... 64
- Referential Integrity ..... 82
- Referential integrity ..... 186
- Referential Integrity (RI) ..... 108
- RefreshWindow ROUTINE ..... 532
- Regenerate button ..... 367
- REGET ..... 735
  - ASCII ..... 565
  - Basic ..... 572
  - Btrieve ..... 583
  - Clarion ..... 598
  - Clipper ..... 606
  - dBaseIII ..... 620
  - dBaseIV ..... 634
  - DOS ..... 646
  - FoxPro ..... 654
  - MSSQL ..... 755
  - ODBC ..... 776
  - Oracle ..... 796
  - Scalable SQL ..... 818, 830
  - TopSpeed ..... 667
- Remake ..... 223, 229
- REMOVE ..... 735
  - ASCII ..... 565
  - Basic ..... 572
  - Btrieve ..... 583
  - Clarion ..... 598
  - Clipper ..... 606
  - dBaseIII ..... 620
  - dBaseIV ..... 634
  - DOS ..... 646
  - FoxPro ..... 654
  - MSSQL ..... 755
  - ODBC ..... 776
  - Oracle ..... 796
  - Scalable SQL ..... 818, 830
  - TopSpeed ..... 667
- REGEX Statement ..... 736
- register add-on template sets ..... 161
- Register allocation ..... 250, 261
- Register-based parameter passing ..... 318
- REGISTERED ..... 499
- Registering the AS400 Driver in Clarion for Windo ..... 714
- Registering the Oracle Driver in Clarion ..... 783
- RegisterTemplateChain ..... 161
- Registry ..... 140
- Registry Options dialog ..... 366
- REGISTRY.TRF ..... 358, 365
- RELATED\_FILE\_TO\_KEY ..... 188
- RELATION ..... 186
- Relation Definition ..... 186
- Relation Definition Group ..... 185
- Relational Database ..... 77
- Relational Database design ..... 108
- relational JOIN operation ..... 110
- Relational Model ..... 77
- relational PROJECT operation ..... 110
- Relations section ..... 185
- RELEASE ..... 99, 480, 735
  - ASCII ..... 565
  - Basic ..... 572
  - Btrieve ..... 583
  - Clarion ..... 598
  - Clipper ..... 606
  - dBaseIII ..... 620
  - dBaseIV ..... 634
  - DOS ..... 646
  - FoxPro ..... 654
  - MSSQL ..... 755
  - ODBC ..... 776
  - Oracle ..... 796
  - Scalable SQL ..... 818, 830
  - TopSpeed ..... 667

|                                         |                         |                                                  |                    |
|-----------------------------------------|-------------------------|--------------------------------------------------|--------------------|
| remove .....                            | 351                     | Resolving Data Types .....                       | 306                |
| RENAME .....                            | 488, 735                | Resolving Naming Conventions .....               | 319                |
| ASCII .....                             | 565                     | response times .....                             | 108                |
| Basic .....                             | 572                     | Restore .....                                    | 273                |
| Btrieve .....                           | 583                     | RESTRICT .....                                   | 186                |
| Clarion .....                           | 598                     | Restrict the action .....                        | 82                 |
| Clipper .....                           | 606                     | result set .....                                 | 110                |
| dBaseIII .....                          | 620                     | RETAIN .....                                     | <b>480</b>         |
| dBaseIV .....                           | 634                     | return attribute .....                           | 493                |
| DOS .....                               | 646                     | return attribute string delimiter position ..... | 501                |
| FoxPro .....                            | 654                     | return current instance number .....             | 496                |
| MSSQL .....                             | 755                     | Return data types .....                          | 314, 316           |
| ODBC .....                              | 776                     | return embed point existence .....               | 494                |
| Oracle .....                            | 796                     | return file existence .....                      | 494                |
| Scalable SQL .....                      | 818, 830                | return from #GROUP .....                         | 462                |
| TopSpeed .....                          | 667                     | return full path                                 |                    |
| Rename .....                            | 156                     | FULLNAME .....                                   | 495                |
| rename .....                            | 352                     | return item exists in list .....                 | 495                |
| Renames .....                           | 242                     | return multi-valued symbol instances .....       | 498                |
| Repeat Procedures= .....                | 134                     | return symbol existence .....                    | 502                |
| REPLACE .....                           | 488, <b>500</b>         | return template registration .....               | 499                |
| Replace .....                           | 156                     | Return values .....                              | 251                |
| replace attribute .....                 | 500                     | Returns address of local variable .....          | 271                |
| replace string special characters ..... | 499, 502                | reusable statement group .....                   | 364, 376           |
| REPORT .....                            | 202, 375, 381           | reuseable .....                                  | 357                |
| Report .....                            | 207                     | REVERSE .....                                    | 451                |
| Report Control Symbols .....            | 519                     | RI .....                                         | 108                |
| Report Formatter .....                  | 145, 390                | RI code in transaction frame .....               | 609, 622, 637, 657 |
| report performance, SQL Drivers .....   | 696                     | right justified .....                            | 342                |
| REQ .....                               | 380, 382, 386, 423, 429 | rightsizing .....                                | 105                |
| REQ attribute .....                     | 540                     | rmdir .....                                      | 353                |
| RequireDictionary= .....                | 134                     | ROLLBACK .....                                   | 730, 735           |
| Reregister if changed= .....            | 140                     | ASCII .....                                      | 565                |
| Reserved words .....                    | 227                     | Basic .....                                      | 572                |
| RESET .....                             | 735                     | Btrieve .....                                    | 583                |
| ASCII .....                             | 565                     | Clarion .....                                    | 598                |
| Basic .....                             | 572                     | Clipper .....                                    | 606                |
| Btrieve .....                           | 583                     | dBaseIII .....                                   | 620                |
| Clarion .....                           | 598                     | dBaseIV .....                                    | 634                |
| Clipper .....                           | 606                     | DOS .....                                        | 646                |
| dBaseIII .....                          | 620                     | FoxPro .....                                     | 654                |
| dBaseIV .....                           | 634                     | MSSQL .....                                      | 755                |
| DOS .....                               | 646                     | ODBC .....                                       | 776                |
| FoxPro .....                            | 654                     | Oracle .....                                     | 796                |
| MSSQL .....                             | 755                     | Scalable SQL .....                               | 818, 830           |
| ODBC .....                              | 776                     | SQL Drivers .....                                | 698                |
| Oracle .....                            | 796                     | TopSpeed .....                                   | 667                |
| Scalable SQL .....                      | 818, 830                | round .....                                      | 335                |
| SQL Drivers .....                       | 688                     | rounding behavior .....                          | 277                |
| TopSpeed .....                          | 667                     | ROUTINE .....                                    | <b>379</b>         |
| RESET Statement .....                   | 736                     | ROWID, Oracle .....                              | 798, 799           |
| Resolving Calling Conventions .....     | 318                     | Run .....                                        | 243                |

- run a Utility template ..... 160
- Run-time error checking ..... 256
- Run-Time Library
  - Local ..... 804
- Run-time stack ..... 255
- runmin= ..... 133
- runtime library functions ..... 328
- runwait= ..... 133
  
- S**
- Safe ..... 234
- SAVE ..... 409
- Save ..... 273
- Save File dialog ..... **443**
- SAVEDIALOG ..... **425, 443**
- SAVESTOREDPROC
  - MSSQL ..... 751
- Scalable SQL
  - Driver Properties ..... 815
  - Driver Strings ..... 814
  - LOGONSCREEN ..... 814
  - Supported File Commands and Functions ..... 818, 830
- Scalable SQL logon ..... 815
- script language ..... 357
- search ..... 340
- SECTION ..... **428, 445, 468**
- section use constraints ..... 401
- SECTION\_ALIGNMENT ..... 297, 300
- Sections ..... 166, 194
- security ..... 726
- security, SQL Drivers ..... 691
- Segment alignment shift ..... 275
- Segment name ..... 249, 253
- SEGMENTS ..... 287, 293
- SELECT
  - SQL Drivers ..... 692
- SELECT COUNT(\*)
  - SQL Drivers ..... 697
- SELECT statement (SQL) ..... 111
- SELECTION ..... **424**
- SEND ..... 735, 796
  - ASCII ..... 565
  - Btrieve ..... 583
  - Clarion ..... 598
  - Clipper ..... 606
  - dBaseIII ..... 620
  - dBaseIV ..... 634
  - DOS ..... 646
  - FoxPro ..... 654
  - MSSQL ..... 755
  - ODBC ..... 776
  - Oracle ..... 796
  - Scalable SQL ..... 818, 830
  - SQL Drivers ..... 693
  - TopSpeed ..... 667
- SETNONNULL ..... 115
- SETNULL ..... 115, 735
  - ASCII ..... 565
  - Basic ..... 572
  - Btrieve ..... 583
  - Clarion ..... 598
- Scalable SQL ..... 818, 830
- SQL Drivers ..... 701
- TopSpeed ..... 667
- SendSQLStatements ..... 743
- Separate source files ..... 34
- SEPARATOR ..... **501**
- Sequential access ..... 87
- Sequential File Access ..... 88
- server
  - multiple ..... 727
- Server-based database engine ..... 105
- SET ..... 88, 89, 735
  - ASCII ..... 565
  - Basic ..... 572
  - Btrieve ..... 583
  - Clarion ..... 598
  - Clipper ..... 606
  - dBaseIII ..... 620
  - dBaseIV ..... 634
  - DOS ..... 646
  - FoxPro ..... 654
  - MSSQL ..... 755
  - ODBC ..... 776
  - Oracle ..... 796
  - Scalable SQL ..... 818, 830
  - TopSpeed ..... 667
- SET / NEXT
  - SQL Drivers ..... 692
- Set threaded= ..... 131
- setdisk ..... 353
- SETNONNULL ..... 735
  - ASCII ..... 565
  - Basic ..... 572
  - Btrieve ..... 583
  - Clarion ..... 598
  - Clipper ..... 606
  - dBaseIII ..... 620
  - dBaseIV ..... 634
  - DOS ..... 646
  - FoxPro ..... 654
  - MSSQL ..... 755
  - ODBC ..... 776
  - Oracle ..... 796
  - Scalable SQL ..... 818, 830
  - SQL Drivers ..... 693
  - TopSpeed ..... 667
- SETNONNULL ..... 115
- SETNULL ..... 115, 735
  - ASCII ..... 565
  - Basic ..... 572
  - Btrieve ..... 583
  - Clarion ..... 598

|                              |                 |                                            |                 |
|------------------------------|-----------------|--------------------------------------------|-----------------|
| Clipper .....                | 606             | Clipper .....                              | 606             |
| dBaseIII .....               | 620             | dBaseIII .....                             | 620             |
| dBaseIV .....                | 634             | dBaseIV .....                              | 634             |
| DOS .....                    | 646             | DOS .....                                  | 646             |
| FoxPro .....                 | 654             | FoxPro .....                               | 654             |
| MSSQL .....                  | 755             | MSSQL .....                                | 755             |
| ODBC .....                   | 776             | ODBC .....                                 | 776             |
| Oracle .....                 | 796             | Oracle .....                               | 796             |
| Scalable SQL .....           | 818, 830        | Scalable SQL .....                         | 818, 830        |
| SQL Drivers .....            | 693             | TopSpeed .....                             | 667             |
| TopSpeed .....               | 667             | SLICE .....                                | 501             |
| SETNULL Statement .....      | 737             | SMALLINT .....                             | 732             |
| Setting Macro Values .....   | 226             | Smart linking .....                        | 264             |
| SETUP .....                  | 783             | SMARTDRV .....                             | 556             |
| Setup Menu .....             | 125             | Software requirements, Oracle .....        | 782             |
| setup prompt symbols .....   | 434             | Software—Developer Requirements .....      | 718             |
| SHARE .....                  | 93, 735         | Software—End User Requirements .....       | 719             |
| ASCII .....                  | 565             | <b>SORT .....</b>                          | <b>424, 430</b> |
| Basic .....                  | 572             | sort order of the result set .....         | 111             |
| Btrieve .....                | 583             | Sort orders .....                          | 87              |
| Clarion .....                | 598             | sorted insert .....                        | 413             |
| Clipper .....                | 606             | Sorting and Collating Sequences .....      | 554             |
| dBaseIII .....               | 620             | Sound .....                                | 328             |
| dBaseIV .....                | 634             | Source .....                               | 217             |
| DOS .....                    | 646             | source code generation time .....          | 360             |
| FoxPro .....                 | 654             | Source file extensions .....               | 304             |
| MSSQL .....                  | 755             | source generation control .....            | 371             |
| ODBC .....                   | 776             | source generation control section .....    | 364             |
| Oracle .....                 | 796             | source generation error .....              | 486             |
| Scalable SQL .....           | 818, 830        | source generation message .....            | 489             |
| TopSpeed .....               | 667             | Specified Driver Could not be Loaded ..... | 740             |
| Shared-access .....          | 93              | SPIN .....                                 | 425, <b>444</b> |
| SHORT .....                  | 306, 732        | SQL .....                                  | 106, 743        |
| SHOW .....                   | <b>381, 385</b> | debugging .....                            | 703             |
| Show on Properties box ..... | 366             | error handling .....                       | 703             |
| showalignbox= .....          | 141, 145        | logging .....                              | 703             |
| showfieldbox= .....          | 141, 145        | tracing .....                              | 703             |
| ShowPriority= .....          | 136             | SQL and ABC Templates .....                | 688             |
| showpropertybox= .....       | 141, 145        | SQL Anywhere performance with ODBC .....   | 772             |
| showtoolbox= .....           | 141, 145        | SQL column names .....                     | 726             |
| Simple Prompts .....         | 212             | SQL databases .....                        | 106             |
| sine .....                   | 330, 336        | SQL Driver Strings .....                   | 705             |
| SINGLE .....                 | 379             | SQL SELECT .....                           | 111, 692        |
| single inheritance .....     | 71              | SQL statements .....                       | 687, 713, 725   |
| SINGLE PRECISION .....       | 732             | SQL statements, ODBC .....                 | 761             |
| Single quotes .....          | 226             | SQL statements, Oracle .....               | 781             |
| Skeleton .....               | 166, 194        | SQL statements, Scalable SQL .....         | 745, 809, 823   |
| SKIP .....                   | 735             | SQLAnywhere .....                          |                 |
| ASCII .....                  | 565             | Driver Properties .....                    | 828             |
| Basic .....                  | 572             | Driver Strings .....                       | 827             |
| Btrieve .....                | 583             | LOGONSCREEN .....                          | 827             |
| Clarion .....                | 598             | SQLAnywhere logon .....                    | 828             |

- SQLChange ..... 743
- SQLException, ODBC ..... 769
- SQLFreeForm ..... 743
- square root ..... 336
- srand ..... 347
- SREAL ..... 306, 732
- Stack frames ..... 261
- Stack provider ..... 720
- Stack size ..... 255
- Stack-Based Data ..... 33
- Stack-based parameter passing ..... 318
- STACK\_COMMIT ..... 297, 299
- STACK\_RESERVE ..... 297, 300
- Stack\_size ..... 257
- stack\_threshold ..... 277
- STACKSIZE ..... 287, 294
- standard customizations ..... 359
- standard window handling code ..... 526
- starting point designs ..... 358
- startX= ..... 130
- startY= ..... 130
- STATUS ..... 735
  - ASCII ..... 565
  - Basic ..... 572
  - Btrieve ..... 583
  - Clarion ..... 598
  - Clipper ..... 606
  - dBaseIII ..... 620
  - dBaseIV ..... 634
  - DOS ..... 646
  - FoxPro ..... 654
  - MSSQL ..... 755
  - ODBC ..... 776
  - Oracle ..... 796
  - Scalable SQL ..... 818, 830
  - TopSpeed ..... 667
- Storage class redeclared ..... 268
- Stored Procedures ..... 108
- stored procedures ..... 114
- Stored Procedures, Oracle ..... 795
- Stored Procedures, SQL Drivers ..... 702
- strcat ..... 347
- strchr ..... 348
- strcmp ..... 347
- strcpy ..... 348
- strcspn ..... 348
- strdup ..... 350
- STREAM ..... 735
  - ASCII ..... 565
  - Basic ..... 572
  - Btrieve ..... 583
  - Clarion ..... 598
  - Clipper ..... 606
  - dBaseIII ..... 620
  - dBaseIV ..... 634
  - DOS ..... 646
  - FoxPro ..... 654
  - MSSQL ..... 755
  - ODBC ..... 776
  - Oracle ..... 796
  - Scalable SQL ..... 818, 830
  - TopSpeed ..... 667
- strequ ..... 347
- strerror ..... 348
- STRING ..... 213, 306, 732
- string ..... 337
- String Functions ..... 347
- string slicing ..... 501
- strlen ..... 348
- strlwr ..... 349
- strncat ..... 350
- strncmp ..... 350
- strncpy ..... 350
- strnicmp ..... 351
- strpbrk ..... 349
- strchr ..... 349
- strrev ..... 350
- strspn ..... 349
- strstr ..... 349
- strtok ..... 349
- Struct ..... 308
- Structured constant ..... 256
- Structured programming ..... 33
- Structured Query Language (SQL) ..... 106
- strupr ..... 350
- Subsections ..... 166, 194
- Substitution ..... 225
- Substitution strings ..... 225
- Super Files, TopSpeed ..... 674
- super-fix multi-value symbols ..... 419
- Supported Commands and Attributes ..... 735
- Supported Data Types ..... 732
- Symbol ..... 213
- symbol ..... 359
- symbol existence ..... 502
- Symbol Hierarchy ..... 505
- Symbol Management Statements ..... 409
- symbols ..... 503
- Symbols Dependent on %Field ..... 509
- Symbols Dependent on %Application ..... 506
- Symbols Dependent on %File ..... 507
- Symbols Dependent on %Formula ..... 520
- Symbols Dependent on %Key ..... 512
- Symbols Dependent on %KeyOrder ..... 512

|                                                   |                    |                                             |                 |
|---------------------------------------------------|--------------------|---------------------------------------------|-----------------|
| Symbols Dependent on %Module .....                | 514                | template registry file .....                | 358             |
| Symbols Dependent on %Procedure .....             | 515                | Template Registry Options .....             | 140             |
| Symbols Dependent on %Relation .....              | 513                | template reset code .....                   | 395             |
| Symbols Dependent on %Report .....                | 519                | Template set .....                          | 371, 373        |
| Symbols Dependent on %ViewFiles .....             | 509                | template set .....                          | 364, 368, 369   |
| Symbols Dependent on %Window .....                | 517                | Template Structure .....                    | 364             |
| SyncWindow ROUTINE .....                          | 532                | Template symbol .....                       | 213             |
| System Macros .....                               | 227                | template symbol (variable) .....            | 364             |
| System path .....                                 | 196                | template variable (symbol) .....            | 359             |
| System Requirements, Oracle .....                 | 781                | terminate the DDE channel .....             | 152             |
| <b>T</b>                                          |                    |                                             |                 |
| TAB .....                                         | 174, 180           | TEXT .....                                  | <b>425, 444</b> |
| ASCII .....                                       | 563                | Text Editor .....                           | 217, 390        |
| Table Import .....                                | 722                | Text editor .....                           | 165, 193, 224   |
| Table Import Wizard .....                         | 721                | Third-party compilers .....                 | 233             |
| Table Names and Aliases .....                     | 724                | THREAD .....                                | 735             |
| Table or View Does Not Exist .....                | 803                | ASCII .....                                 | 565             |
| TABs .....                                        | 127                | Basic .....                                 | 572             |
| tangent .....                                     | 331, 336           | Btrieve .....                               | 583             |
| Target file .....                                 | 233                | Clarion .....                               | 598             |
| target language .....                             | 357                | Clipper .....                               | 606             |
| target language source code .....                 | 359                | dBaseIII .....                              | 620             |
| target language statement .....                   | 364                | dBaseIV .....                               | 634             |
| Target operating system .....                     | 230                | DOS .....                                   | 646             |
| targets for control and extension templates ..... | 361                | FoxPro .....                                | 654             |
| tbposh= .....                                     | 141, 145           | MSSQL .....                                 | 755             |
| tbposw= .....                                     | 141, 145           | ODBC .....                                  | 776             |
| tbposx= .....                                     | 141, 145           | Oracle .....                                | 796             |
| tbposy= .....                                     | 141, 145           | Scalable SQL .....                          | 818, 830        |
| TCF .....                                         |                    | TopSpeed .....                              | 667             |
| TopSpeed .....                                    | 666, 673           | TIME .....                                  | 306, 312, 732   |
| TCP/IP host .....                                 | 786, 789           | Oracle .....                                | 798             |
| teaching example .....                            | 742                | time .....                                  | 332             |
| Template class .....                              | 205, 219           | TIME, Nulls and ODBC .....                  | 772             |
| template class .....                              | 364                | Time stamp .....                            | 242             |
| template code comments .....                      | 483                | TIMER attribute .....                       | 549             |
| template code files .....                         | 365                | TIMES .....                                 | 214, <b>454</b> |
| Template Code Sections .....                      | 369                | TIMESTAMP .....                             | 732             |
| template code sections .....                      | 364                | Title .....                                 | 176, 182        |
| template help file .....                          | 487                | tolower .....                               | 345             |
| template initialization code .....                | 394                | Too many arguments .....                    | 251             |
| template language procedure or function .....     | 364                | TOOOPTION .....                             | 190             |
| Template Language statements .....                |                    | TOOLTIP .....                               | 176, 183        |
| placed in embed points .....                      | 217                | Topspeed Database Copy Utility .....        | 682             |
| Template name .....                               | 213                | TopSpeed Database Recovery Utility .....    | 677             |
| Template Prompts .....                            | 362                | TopSpeed Driver .....                       |                 |
| Template prompts .....                            | 212                | full key build .....                        | 665             |
| Template Properties dialog .....                  | 367                | TopSpeed File Driver .....                  |                 |
| Template Registry .....                           | 365, 366, 369, 373 | Supported File Commands and Functions ..... | 667             |
| template registry .....                           | 368                | TopSpeed Files .....                        |                 |
|                                                   |                    | Data Compression .....                      | 672             |
|                                                   |                    | Estimating File Size .....                  | 672             |
|                                                   |                    | MEMOs versus STRINGS .....                  | 668, 672        |

- Transaction Processing ..... 673
  - TopSpeed files ..... 663
    - Data Types ..... 664
    - Driver Strings ..... 665
    - File Specifications ..... 664
    - Key Rebuild ..... 683, 684
    - Open access modes ..... 670
    - SEND functions ..... 665
    - SHARE ..... 670
    - Storing multiple Tables in a single file ..... 674
  - Touch ..... 242
  - toupper ..... 344
  - TPSBT error ..... 670, 677
  - TPSCOPY ..... 682, 684
  - TPSFIX ..... 679
    - example file ..... 677
  - TR\$, Clarion Driver ..... 600
  - Tracing ..... 263
  - Transaction Control File ..... 663
  - transaction framing ..... 730
  - transaction processing ..... 730
  - Transaction Processing, TopSpeed Files ..... 673
  - TranslateControl= ..... 135
  - TREE ..... **389**
    - tree diagram of procedure calls ..... 378
  - Triggers ..... 108
  - TRUSTEDCONNECTION
    - MSSQL ..... 752
  - Tsexemod ..... 244
  - Tsimplib ..... 235
  - Type inconsistency ..... 316
  - TYPEMODE ..... 177, 183
- ## U
- ULONG ..... 306
  - Unconditional control structures ..... 358
  - Undefined symbols warning ..... 264
  - Unexpected End of SQL Command ..... 803
  - Unexpected text in preprocessor command ..... 268
  - unfix a multi-value symbol ..... 422
  - Uninterruptable Power Supply ..... 556
  - union operation ..... 413
  - UNIQUE ..... 212, 409, **424**
  - Unique KEY ..... 78
  - Unique Keys, SQL Drivers ..... 688
  - Unknown #pragma ..... 269
  - Unknown compiler ..... 304
  - UNLOCK ..... 101, 735
    - ASCII ..... 565
    - Basic ..... 572
    - Btrieve ..... 583
  - Clarion ..... 598
  - Clipper ..... 606
  - dBaseIII ..... 620
  - dBaseIV ..... 634
  - DOS ..... 646
  - FoxPro ..... 654
  - MSSQL ..... 755
  - ODBC ..... 776
  - Oracle ..... 796
  - Scalable SQL ..... 818, 830
  - TopSpeed ..... 667
  - UNQUOTE ..... **502**
  - Unsupported Function ..... 739
  - UNTIL ..... 454
  - unused columns, SQL Drivers ..... 695
  - UPDATE ..... 186
  - Update Template Chain= ..... 140
  - update the registry ..... 367
  - UPPER ..... 177, 183
  - uppercase ..... 342
  - UPS ..... 556
  - USAGE ..... 171
  - USEINNERJOIN
    - ODBC ..... 772
  - User Defined Applications ..... 123
  - User Interface ..... 237
  - User Menus ..... 124
  - User Options ..... 363
  - user-defined symbol ..... 409
  - User-defined symbols ..... 371
  - Username ..... 726, 787, 789
  - Username, ODBC ..... 767
  - Username, SQL Drivers ..... 691
  - Username, SQLAnywhere ..... 824
  - USETRANSACTIONS ..... 730, 737
  - USHORT ..... 306
  - utility execution section ..... 365, 378
  - Utility Templates ..... 206
- ## V
- VALID ..... 175, 181
  - validate prompt input ..... 427
  - validation, data ..... 109
  - Validation Statements ..... 423
  - Validity Checks ..... 176, 182
  - VALUE ..... **424**
  - Value of constant is out of range ..... 271
  - Value of escape sequence is too large ..... 271
  - Value-parameters ..... 376
  - value-parameters ..... 459, 460, 461, 492, 497
  - VALUES ..... 182

|                                        |                         |                                 |                                        |
|----------------------------------------|-------------------------|---------------------------------|----------------------------------------|
| VARCHAR .....                          | 732                     | WHEN .....                      | 214, 216                               |
| VARCHAR2, Oracle .....                 | 798                     | WHENACCEPTED .....              | <b>424, 430</b>                        |
| VAREXISTS .....                        | <b>502</b>              | WHERE .....                     | 388, 391, 401, 429, 433, 445, 448, 451 |
| Variable .....                         | 225                     | ODBC .....                      | 769                                    |
| Variable declared but never used ..... | 270                     | Oracle .....                    | 793                                    |
| Variable file names, Scalable SQL      |                         | SQL Drivers .....               | 706                                    |
| 749, 756, 811, 819, 826, 831           |                         | WHERE clause (SQL) .....        | 111, 114                               |
| Variable number of arguments .....     | 251                     | WHILE .....                     | 454                                    |
| Variable parameter lists .....         | 252                     | Win .....                       | 230                                    |
| variable username .....                | 726                     | WIN32 .....                     | <b>480</b>                             |
| Variable-parameters .....              | 377                     | Win32 .....                     | 230                                    |
| variable-parameters .....              | 459, 460, 461, 492, 497 | WINDOW .....                    | 375, 381                               |
| variables .....                        | 357, 503                | Window .....                    | 206                                    |
| Variant record .....                   | 254                     | WINDOW attribute .....          | 544                                    |
| VERIFYVIASELECT                        |                         | Window Control Symbols .....    | 517                                    |
| ODBC .....                             | 772                     | Window Formatter .....          | 141, 361, 390                          |
| VERSION .....                          | 168, 196                | window handling procedure ..... | 525                                    |
| Version number .....                   | 168, 196                | Window procedure template ..... | 525                                    |
| Versions .....                         | 223                     | WINDOWAPI .....                 | 288, 298                               |
| VERTICALSPACE .....                    | 174, 180                | Windows 95 .....                | 303                                    |
| VID .....                              | 262                     | Windows API .....               | 327                                    |
| VIEW, dynamic JOINS .....              | 710                     | Windows help file .....         | 196                                    |
| VIEW structure .....                   | 110                     | Windows NT .....                | 303                                    |
| ViewPrint .....                        | 744                     | Windows programs .....          | 244                                    |
| VIRTUAL attribute .....                | 73                      | windows95digs= .....            | 120                                    |
| Virtual Method Table .....             | 61                      | WIZARD .....                    | <b>378</b>                             |
| Virtual Methods .....                  | 61, 73                  | wizard .....                    | <b>375</b>                             |
| Visual Interactive Debugger .....      | 262                     | Wizards .....                   | 170, 172, 173, 179, 180, 184           |
| VMT .....                              | 61                      | Working directory .....         | 239                                    |
| Volatile variables .....               | 254                     | WRAP .....                      | <b>381</b>                             |
|                                        |                         | Write Only .....                | 93                                     |
| <b>W</b>                               |                         |                                 |                                        |
| WAIT .....                             | <b>479</b>              | <b>X</b>                        |                                        |
| wallpaper= .....                       | 120                     | XBase commands                  |                                        |
| wallpapermode= .....                   | 120                     | Clipper .....                   | 611                                    |
| Warnings .....                         | 266                     | dBaseIII .....                  | 624                                    |
| WATCH .....                            | 735                     | dBaseIV .....                   | 639                                    |
| ASCII .....                            | 565                     | XBase file format               |                                        |
| Basic .....                            | 572                     | dBaseIII .....                  | 615                                    |
| Btrieve .....                          | 583                     | Xtrieve .....                   | 580                                    |
| Clarion .....                          | 598                     | <b>Y</b>                        |                                        |
| Clipper .....                          | 606                     | year .....                      | 337                                    |
| dBaseIII .....                         | 620                     | <b>Z</b>                        |                                        |
| dBaseIV .....                          | 634                     | zero_divide .....               | 276                                    |
| DOS .....                              | 646                     | ZEROISNULL                      |                                        |
| FoxPro .....                           | 654                     | ODBC .....                      | 772                                    |
| MSSQL .....                            | 755                     | ZONED DECIMAL .....             | 732                                    |
| ODBC .....                             | 769, 776                |                                 |                                        |
| Oracle .....                           | 796                     |                                 |                                        |
| Scalable SQL .....                     | 818, 830                |                                 |                                        |
| TopSpeed .....                         | 667                     |                                 |                                        |