

Language Reference Manual



Clarion **6**

SoftVelocity

COPYRIGHT 1994-2003 SoftVelocity Incorporated. All rights reserved.

This publication is protected by copyright and all rights are reserved by SoftVelocity Incorporated. 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 SoftVelocity Incorporated.

This publication supports Clarion. It is possible that it may contain technical or typographical errors. SoftVelocity Incorporated provides this publication "as is," without warranty of any kind, either expressed or implied.

SoftVelocity Incorporated

2769 East Atlantic Blvd.
Pompano Beach, Florida 33062
(954) 785-4555
www.softvelocity.com

Trademark Acknowledgements:

SoftVelocity is a trademark of SoftVelocity Incorporated.

Clarion™ is a trademark of SoftVelocity Incorporated.

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.

Printed in the United States of America (1103)

Contents:

1 - Introduction	23
Introduction--The Language Reference Manual	23
The Language Reference Manual	24
Language Reference -- Chapter Organization	24
Documentation Conventions and Symbols	26
Reference Item Format	27
KEYWORD (short description of intended use)	28
Clarion Conventions.....	29
Standard Date	29
Standard Time	30
Clarion Keycodes	31
2 - Program Source Code Format	33
Statement Format	33
Declaration and Statement Labels.....	34
Structure Termination	35
Field Qualification	36
Reserved Words.....	38
Special Characters	39
Program Format.....	41
PROGRAM (declare a program)	41
MEMBER (identify member source file)	43
MAP (declare PROCEDURE prototypes)	45
MODULE (specify MEMBER source file)	47
PROCEDURE (define a procedure)	48
CODE (begin executable statements).....	51
DATA (begin routine local data section).....	52
ROUTINE (declare local subroutine).....	53
END (terminate a structure)	55
Statement Execution Sequence	56
PROCEDURE Calls.....	57
PROCEDURE Prototypes	58
Prototype Syntax	58
Prototype Parameter Lists - General Syntax.....	61
PROCEDURE Return Types.....	69
Prototype Attributes	71
C, PASCAL (parameter passing conventions)	71
DERIVED (prevent function overloading).....	72
DLL (set procedure defined externally in .DLL).....	73
NAME (set prototype's external name)	74
PRIVATE (set procedure private to a CLASS or module).....	75
PROC (set function called as procedure without warnings).....	76
PROTECTED (set procedure private to a CLASS or derived CLASS)	77

RAW (pass address only).....	78
REPLACE (set replacement constructor or destructor)	79
TYPE (specify PROCEDURE type definition)	81
VIRTUAL (set virtual method)	82
Procedure Overloading	83
Rules for Procedure Overloading	84
Name Mangling and C++ Compatibility.....	86
Compiler Directives.....	87
ASSERT (set assumption for debugging)	87
BEGIN (define code structure)	89
COMPILE (specify source to compile)	90
INCLUDE (compile code in another file)	92
EQUATE (assign label)	93
ITEMIZE (enumeration data structure).....	94
OMIT (specify source not to be compiled)	96
ONCE (prevent duplicate included data).....	98
SECTION (specify source code section).....	99
SIZE (memory size in bytes)	100

3 - Variable Declarations

103

Simple Data Types.....	103
BYTE (one-byte unsigned integer)	103
SHORT (two-byte signed integer)	104
USHORT (two-byte unsigned integer).....	106
LONG (four-byte signed integer)	108
ULONG (four-byte unsigned integer)	110
SIGNED (signed integer).....	112
UNSIGNED (unsigned integer)	113
SREAL (four-byte signed floating point).....	114
REAL (eight-byte signed floating point).....	116
BFLOAT4 (four-byte signed floating point).....	118
BFLOAT8 (eight-byte signed floating point)	120
DECIMAL (signed packed decimal)	122
PDECIMAL (signed packed decimal)	124
STRING (fixed-length string)	126
ASTRING (atomic string).....	128
CSTRING (fixed-length null terminated string).....	130
PSTRING (embedded length-byte string)	133
Implicit String Arrays and String Slicing	136
DATE (four-byte date)	137
TIME (four-byte time).....	139
Special Data Types.....	141
ANY (any simple data type).....	141
LIKE (inherited data type).....	144
Implicit Variables	147
Reference Variables	149
Data Declarations and Memory Allocation.....	153

Global, Local, Static, and Dynamic	153
Data Declaration Sections	154
Variable Size Declarations	155
NEW (allocate heap memory)	156
DISPOSE (de-allocate heap memory)	157
Picture Tokens	159
Numeric and Currency Pictures	159
Scientific Notation Pictures	162
String Pictures	163
Date Pictures	164
Time Pictures	166
Pattern Pictures	167
Key-in Template Pictures	168
4 - Entity Declarations	171
Complex Data Structures	171
GROUP (compound data structure)	171
CLASS (object declaration)	174
INTERFACE (class behavior definition)	184
File Structures	186
FILE (declare a data file structure)	186
INDEX (declare static file access index)	189
KEY (declare dynamic file access index)	190
MEMO (declare a text field)	192
BLOB (declare a variable-length field)	193
RECORD (declare record structure)	196
Null Data Processing	197
FILE Structure Properties	198
Environment Files	209
View Structures	212
VIEW (declare a "virtual" file)	212
PROJECT (set view fields)	216
JOIN (declare a "join" operation)	217
Queue Structures	220
QUEUE (declare a memory QUEUE structure)	220
Additional QUEUE Considerations	223
5 - Declaration Attributes	231
Variable and Entity Attributes	231
AUTO (uninitialized local variable)	231
BINARY (memo contains binary data)	232
BINDABLE (set runtime expression string variables)	233
CREATE (allow data file creation)	234
DIM (set array dimensions)	235
DLL (set variable defined externally in .DLL)	237
DRIVER (specify data file type)	239

DUP (allow duplicate KEY entries).....	240
ENCRYPT (encrypt data file)	241
EXTERNAL (set defined externally)	242
FILTER (set view filter expression)	245
IMPLEMENTS(add methods to a CLASS).....	248
INTERFACE (class behavior definition)	249
INNER (set inner join operation)	251
LINK (specify CLASS link into project).....	252
MODULE (specify CLASS member source file).....	253
NAME (set external name)	254
NOCASE (case insensitive KEY or INDEX).....	258
OEM (set international string support).....	259
OPT (exclude null KEY or INDEX entries)	260
ORDER (set view sort order expression)	261
OVER (set shared memory location).....	263
OWNER (declare password for data encryption)	264
PRE (set label prefix).....	265
PRIMARY (set relational primary key).....	267
PRIVATE (set variable private to a CLASS module).....	268
PROTECTED (set variable private to a CLASS or derived CLASS).....	269
RECLAIM (reuse deleted record space)	270
STATIC (set allocate static memory).....	271
THREAD (set thread-specific memory allocation).....	272
TYPE (type definition)	274
6 - Windows	277
Window Structures	277
APPLICATION (declare an MDI frame window)	277
WINDOW (declare a dialog window).....	283
MENUBAR (declare a pulldown menu).....	290
TOOLBAR (declare a tool bar)	294
Window Overview	297
Window Controls and Input Focus	298
Field Equate Labels.....	299
Graphics Overview.....	301
Current Target	301
Graphics Coordinates.....	302
7 - Reports	303
Report Structures	303
REPORT (declare a report structure).....	303
BREAK (declare group break structure).....	307
DETAIL (report detail line structure).....	308
FOOTER (page or group footer structure)	310
FORM (page layout structure).....	312
HEADER (page or group header structure)	313

Printer Control Properties	315
Page Overflow	318
8 - Controls	321
Control Declarations	321
BOX (declare a box control)	321
BUTTON (declare a pushbutton control)	323
CHECK (declare a checkbox control)	326
COMBO (declare an entry/list control)	329
ELLIPSE (declare an ellipse control)	334
ENTRY (declare a data entry control)	336
GROUP (declare a group of controls)	340
IMAGE (declare a graphic image control)	343
ITEM (declare a menu item)	345
LINE (declare a line control)	348
LIST (declare a window list control)	350
MENU (declare a menu box)	355
OLE (declare a window OLE or .OCX container control)	357
OPTION (declare a set of RADIO controls)	361
PANEL (declare a panel control)	364
PROMPT (declare a prompt control)	365
PROGRESS (declare a progress control)	367
RADIO (declare a radio button control)	369
REGION (declare a window region control)	372
SHEET (declare a group of TAB controls)	374
SPIN (declare a spinning list control)	377
STRING (declare a string control)	380
TAB (declare a page of a SHEET control)	383
TEXT (declare a multi-line text control)	385
9 - Window and Report Attributes	389
Attribute Property Equates	389
Window and Report Attributes	391
ABSOLUTE (set fixed-position printing)	391
ALONE (set to print without page header, footer, or form)	392
ALRT (set window "hot" keys)	393
ANGLE (set control display or print angle)	395
AT (set position and size)	396
AUTO (set USE variable automatic re-display)	399
AUTOSIZE (set OLE object resizing)	400
AVE (set report total average)	401
BEVEL (set 3-D effect border)	402
BOXED (set controls group border)	403
CAP, UPR (set case)	404
CENTER (set centered window position)	405
CENTERED (set centered image)	406

CHECK (set on/off ITEM)	407
CLIP (set OLE object clipping)	408
CNT (set total count)	409
COLOR (set color)	410
COLUMN (set list box highlight bar)	412
EXTEND (set document formatting)	413
COMPATIBILITY (set OLE control compatibility)	414
CREATE (create OLE control object)	415
CURSOR (set mouse cursor type)	416
DEFAULT (set enter key button)	417
DELAY (set repeat button delay)	418
DISABLE (set control dimmed at open)	419
DOCK (set dockable toolbox window)	420
DOCKED (set dockable toolbox window docked at open)	421
DOCUMENT (create OLE control object from file)	422
DOUBLE, NOFRAME, RESIZE (set window border)	423
DRAGID (set drag-and-drop host signatures)	424
DROP (set list box behavior)	425
DROPID (set drag-and-drop target signatures)	426
FILL (set fill color)	427
FIRST, LAST (set MENU or ITEM position)	428
FLAT (set flat control)	429
FONT (set default font)	430
FORMAT (set LIST or COMBO layout)	433
FORMAT() Runtime Properties	438
FORMAT() Style Properties	441
FORMAT() Other List Box Properties	443
FORMAT() List Box Mouse Click Properties	445
FROM (set listbox data source)	448
FULL (set full-screen)	450
GRAY (set 3-D look background)	451
GRID (set list grid-line display color)	452
HIDE (set control hidden)	453
HLP (set on-line help identifier)	454
HSCROLL, VSCROLL, HVSCROLL (set scroll bars)	455
ICON (set icon)	456
ICONIZE (set window open as icon)	458
IMM (set immediate event notification)	459
INS, OVR (set typing mode)	461
JOIN (set joined TAB scroll buttons)	462
KEY (set execution keycode)	463
LANDSCAPE (set page orientation)	464
LAYOUT (set window orientation)	465
LEFT, RIGHT, ABOVE, BELOW (set TAB position)	466
LEFT, RIGHT, CENTER, DECIMAL (set justification)	467
LINEWIDTH (set line thickness)	469
LINK (create OLE control link to object from file)	470

MARK (set multiple selection mode)	471
MASK (set pattern editing data entry)	472
MAX (set maximize control or total maximum).....	473
MAXIMIZE (set window open maximized)	475
MDI (set MDI child window).....	476
MIN (set total minimum)	477
MODAL (set system modal window)	478
MSG (set status bar message).....	479
NOBAR (set no highlight bar).....	480
NOCASE (case insensitive report BREAK).....	481
NOMERGE (set merging behavior).....	482
NOSHEET (set "floating" TABs).....	484
OPEN (open OLE control object from file)	485
PAGE (set page total reset)	486
PAGEAFTER (set page break after)	487
PAGEBEFORE (set page break first).....	488
PAGENO (set page number print).....	489
PALETTE (set number of hardware colors)	490
PAPER (set report paper size)	491
PASSWORD (set data non-display).....	492
PREVIEW (set report output to metafiles).....	493
RANGE (set range limits)	495
READONLY (set display-only)	496
REPEAT (set repeat button rate)	497
REQ (set required entry)	498
RESET (set total reset)	499
RESIZE (set variable height TEXT control).....	500
RIGHT (set MENU position)	501
ROUND (set round-cornered BOX).....	502
RTF (declare TEXT control as RichEdit).....	503
SCROLL (set scrolling control).....	504
SEPARATOR (set separator line ITEM).....	505
SINGLE (set TEXT for single line entry).....	506
SKIP (set Tab key skip or conditional print control)	507
SMOOTH (set smooth progress bar increments).....	508
SPREAD (set evenly spaced TAB controls).....	508
STATUS (set status bar)	509
STD (set standard behavior)	511
STEP (set SPIN increment).....	512
STRETCH (set OLE object stretching).....	513
SUM (set total).....	514
SYSTEM (set system menu)	515
TALLY (set total calculation times).....	516
THOUS, MM, POINTS (set report coordinate measure).....	517
TILED (set tiled image).....	518
TIMER (set periodic event).....	519
TIP (set "balloon help" text)	520

TOGETHER (set to print on a single page).....	521
TOOLBOX (set toolbox window behavior)	522
TRN (set transparent control).....	524
UP, DOWN (set TAB text orientation)	525
USE (set field equate label or control update variable).....	526
VALUE (set RADIO or CHECK control USE variable assignment).....	529
VCR (set VCR control)	530
VERTICAL (set vertical progress bar display).....	531
WALLPAPER (set background image).....	532
WITHNEXT (set widow elimination)	533
WITHPRIOR (set orphan elimination)	534
WIZARD (set "tabless" SHEET control)	535
ZOOM (set OLE object zooming).....	536
10 - Expressions	537
Overview	537
Expression Evaluation	537
Operators	538
Arithmetic Operators.....	538
The Concatenation Operator	539
Logical Operators	540
Constants	541
Numeric Constants	541
String Constants	542
Types of Expressions.....	543
Numeric Expressions.....	543
String Expressions.....	544
Logical Expressions.....	545
Property Expressions	546
Runtime Expression Evaluation	549
BIND (declare runtime expression string variable).....	550
BINDEXPRESSION(declare runtime expression).....	552
EVALUATE (return runtime expression string result).....	553
POPBIND (restore runtime expression string name space)	555
PUSHBIND (save runtime expression string name space).....	556
UNBIND (free runtime expression string variable)	557
11 - Assignments	559
Simple Assignment Statements	559
Operating Assignments	560
Deep Assignment	561
Reference Assignments	563
CLEAR (clear a variable).....	565
Data Type Conversion Rules	566
Base Types.....	567
BCD Operations and Procedures.....	569

Type Conversion and Intermediate Results	570
Simple Assignment Data Type Conversion.....	571
12 - Execution Control	579
Control Structures	579
ACCEPT (the event processor).....	579
CASE (selective execution structure).....	582
EXECUTE (statement execution structure).....	584
IF (conditional execution structure)	586
LOOP (iteration structure)	588
Execution Control Statements	591
BREAK (immediately leave loop)	591
CYCLE (go to top of loop)	592
DO (call a ROUTINE)	594
EXIT (leave a ROUTINE)	595
GOTO (go to a label)	596
RETURN (return to caller)	597
13 - Built-in Procedures	599
Procedure Listing by Function	599
ABS (return absolute value).....	608
ACCEPTED (return control just completed)	609
ACOS (return arccosine).....	610
ADD (add an entry)	611
ADDRESS (return memory address).....	615
AGE (return age from base date)	616
ALERT (set event generation key).....	617
ALIAS (set alternate keycode)	619
ALL (return repeated characters).....	620
APPEND (add a new file record)	621
ARC (draw an arc of an ellipse).....	622
ASK (get one keystroke)	623
ASIN (return arcsine)	624
ATAN (return arctangent).....	625
BAND (return bitwise AND).....	626
BEEP (sound tone on speaker)	627
BLANK (erase graphics)	628
BOR (return bitwise OR)	629
BOX (draw a rectangle)	630
BSHIFT (return shifted bits)	631
BUFFER (set record paging)	632
BUILD (build keys and indexes).....	634
BXOR (return bitwise exclusive OR).....	637
BYTES (return size in bytes).....	638
CALL (call procedure from a DLL)	640
CALLBACK (register or unregister a FileCallbackInterface)	641

CENTER (return centered string)	644
CHAIN (execute another program)	645
CHANGE (change control field value)	646
CHANGES (return changed queue)	647
CHOICE (return relative item position)	648
CHOOSE (return chosen value)	649
CHORD (draw a section of an ellipse)	651
CHR (return character from ASCII)	652
CLIP (return string without trailing spaces)	653
CLIPBOARD (return windows clipboard contents)	654
CLOCK (return system time)	656
CLONE(duplicate existing control)	657
CLOSE (close a data structure)	659
COLORIALOG (return chosen color)	661
COMMAND (return command line)	662
COMMIT (terminate successful transaction)	663
CONTENTS (return contents of USE variable)	664
CONVERTANSITOOEM (convert ANSI strings to ASCII)	665
CONVERTOEMTOANSI (convert ASCII strings to ANSI)	666
COPY (copy a file)	667
COS (return cosine)	668
CREATE (create an empty data file)	669
CREATE (return new control created)	670
DATE (return standard date)	673
DAY (return day of month)	674
DEFORMAT (return unformatted numbers from string)	675
DELETE (delete a record)	676
DELETEREG (remove a value or key from Windows registry)	679
DESTROY (remove a control or file)	680
DIRECTORY (get file directory)	682
DISABLE (dim a control)	685
DISPLAY (write USE variables to screen)	686
DRAGID (return matching drag-and-drop signature)	687
DROPID (return drag-and-drop string)	689
DUPLICATE (check for duplicate key entries)	691
ELLIPSE (draw an ellipse)	692
EMPTY (empty a data file)	693
ENABLE (re-activate dimmed control)	694
ENDPAGE (force page overflow)	695
ERASE (clear screen control and USE variables)	697
ERROR (return error message)	698
ERRORCODE (return error code number)	699
ERRORFILE (return error filename)	700
EVENT (return event number)	701
EXISTS (return file existence)	702
FIELD (return control with focus)	703
FILEIALOG (return chosen file)	704

FILEDIALOGA (extended file dialog)	707
FILEERROR (return file driver error message).....	711
FILEERRORCODE (return file driver error code number).....	712
FIRSTFIELD (return first window control)	713
FLUSH (flush buffers)	714
FOCUS (return control with focus).....	715
FONTDIALOG (return chosen font)	716
FONTDIALOGA (return chosen font and character set).....	717
FORMAT (return formatted numbers into a picture)	718
FORWARDKEY (pass keystrokes to control)	719
FREE (delete all entries)	720
FREESTATE (free resources)	721
GET (read a record or entry).....	722
GETFONT (get font information).....	726
GETINI (return INI file entry)	727
GETNULLS(get the NULL state of a table).....	728
GETPOSITION (get control position).....	729
GETREG(get Windows registry entry)	730
GETSTATE (return current state of data file)	731
HALT (exit program)	732
HELP (help window access)	733
HIDE (blank a control).....	734
HOLD (exclusive record access)	735
IDLE (arm periodic procedure).....	737
IMAGE (draw a graphic image).....	738
INCOMPLETE (return empty REQ control)	739
INLIST (return entry in list).....	740
INRANGE (check number within range)	741
INSTANCE (return variable's thread instance address)	742
INSTRING (return substring position).....	744
INT (truncate fraction)	746
ISALPHA (return alphabetic character)	747
ISLOWER (return lower case character)	748
ISSTRING (return field string type or not).....	749
ISUPPER (return upper case character)	750
KEYBOARD (return keystroke waiting)	751
KEYCHAR (return ASCII code).....	752
KEYCODE (return last keycode).....	753
KEYSTATE (return keyboard status).....	754
LASTFIELD (return last window control)	755
LEFT (return left justified string).....	756
LEN (return length of string).....	757
LINE (draw a straight line)	758
LOCALE (load environment file)	759
LOCK (exclusive file access)	760
LOCKTHREAD (re-lock the current execution thread)	761
LOG10 (return base 10 logarithm).....	762

LOGE (return natural logarithm)	763
LOGOUT (begin transaction)	764
LONGPATH (return long filename)	766
LOWER (return lower case)	767
MATCH (return matching values)	768
MAXIMUM (return maximum subscript value)	772
MESSAGE (return message box response)	773
MONTH (return month of date)	776
MOUSEX (return mouse horizontal position)	777
MOUSEY (return mouse vertical position)	778
NAME (return file name)	779
NEXT (read next record in sequence)	780
NOMEMO (read file record without reading memo)	782
NOTIFY (send safe information to a receiver thread)	783
NOTIFICATION (receive information from sender thread)	784
NULL (return null file field)	786
NUMERIC (return numeric string)	788
OMITTED (return omitted parameters)	789
OPEN (open a data structure)	791
PACK (remove deleted records)	795
PATH (return current directory)	796
PEEK (read memory address)	797
PENCOLOR (return line draw color)	798
PENSTYLE (return line draw style)	799
PENWIDTH (return line draw thickness)	800
PIE (draw a pie chart)	801
POINTER (return last queue entry position)	803
POKE (write to memory address)	804
POLYGON (draw a multi-sided figure)	805
POPUP (return popup menu selection)	806
POSITION (return record sequence position)	808
POST (post user-defined event)	811
PRAGMA (control pragma settings from source)	812
PRESS (put characters in the buffer)	814
PRESSKEY (put a keystroke in the buffer)	815
PREVIOUS (read previous view record in sequence)	816
PRINT (print a report structure)	818
PRINTERDIALOG (return chosen printer)	819
POPERRORS (return error information)	820
PUSHERRORS (write error information)	822
PUT (re-write record)	824
PUTINI (set INI file entry)	828
PUTREG (write value to Windows registry)	829
QUOTE (replace string special characters)	830
RANDOM (return random number)	831
RECORDS (return number of rows in data set)	832
REGISTER (register event handler)	834

REJECTCODE (return reject code number).....	836
REGET (re-get record).....	837
RELEASE (release a held record).....	839
REMOVE (erase a file).....	840
RENAME (change file directory name).....	841
RESET (reset record sequence position).....	842
RESTORESTATE (restore state of data file).....	845
RESUME (resume thread execution).....	846
RIGHT (return right justified string).....	847
ROLLBACK (terminate unsuccessful transaction).....	848
ROUND (return rounded number).....	849
ROUNDBOX (draw a box with round corners).....	850
RUN (execute command).....	851
RUNCODE (return program exit code).....	853
SELECT (select next control to process).....	854
SELECTED (return control that has received focus).....	856
SEND (send message to file driver).....	857
SET (initiate sequential file processing).....	858
SET3DLOOK (set 3D window look).....	862
SETCLIPBOARD (set windows clipboard contents).....	863
SETCLOCK (set system time).....	864
SETCOMMAND (set command line parameters).....	865
SETCURSOR (set temporary mouse cursor).....	866
SETDROPID (set DROPID return string).....	867
SETFONT (specify font).....	868
SETKEYCHAR (specify ASCII code).....	869
SETKEYCODE (specify keycode).....	870
SETNONULL (set file field non-null).....	871
SETNULL (set file field null).....	873
SETNULLS(set the null state of columns).....	875
SETPATH (change current drive and directory).....	876
SETPENCOLOR (set line draw color).....	877
SETPENSTYLE (set line draw style).....	878
SETPENWIDTH (set line draw thickness).....	879
SETPOSITION (specify new control position).....	880
SETTARGET (set current window or report).....	881
SETTODAY (set system date).....	883
SHORTPATH (return short filename).....	884
SHOW (write to screen).....	885
SHUTDOWN (arm termination procedure).....	886
SIN (return sine).....	887
SKIP (bypass records in sequence).....	888
SORT (sort queue entries).....	890
SQL (use SQL code).....	892
SQLCALLBACK (register or unregister a SQLCallbackInterface).....	893
SQRT (return square root).....	896
START (return new execution thread).....	897

STATUS (return file or window/report status)	899
STOP (suspend program execution)	901
STREAM (enable operating system buffering)	902
STRPOS (return matching value position).....	903
SUB (return substring of string)	907
SUSPEND (suspend thread execution).....	909
TAN (return tangent).....	910
THREAD (return current execution thread)	911
THREADLOCKED (returns current execution thread locked state)	912
TIE(associate a string value to an ASTRING)	913
TIED(retrieves a value associated with an ASTRING)	914
TODAY (return system date)	915
TYPE (write string to screen).....	916
UNHIDE (show hidden control).....	917
UNLOAD (remove a CALLED DLL from memory).....	918
UNLOCK (unlock a locked data file)	919
UNLOCKTHREAD (unlock the current execution thread)	920
UNQUOTE (remove string special characters).....	921
UNREGISTER (unregister event handler)	922
UNTIE(disassociate a string value from an ASTRING)	923
UPDATE (write from screen to USE variables)	924
UPPER (return upper case)	925
VAL (return ASCII value)	926
WATCH (automatic concurrency check).....	927
WHAT (return field from group).....	928
WHERE (return field position in group).....	929
WHO (return field name from group)	930
YEAR (return year of date)	931
YIELD (allow event processing).....	932

Appendix A - DDE, OLE, and .OCX **935**

Dynamic Data Exchange	935
DDE Overview	935
DDE Events	936
DDE Procedures	938
DDEACKNOWLEDGE (send acknowledgement from DDE server)	938
DDEAPP (return server application).....	940
DDECHANNEL (return DDE channel number)	941
DDECLIENT (return DDE client channel).....	942
DDECLOSE (terminate DDE server link)	943
DDEEXECUTE (send command to DDE server)	944
DDEITEM (return server item).....	946
DDEPOKE (send unsolicited data to DDE server)	947
DDEQUERY (return registered DDE servers).....	949
DDEREAD (get data from DDE server).....	950
DDESERVER (return DDE server channel)	952
DDETOPIC (return server topic)	953

DDEVALUE (return data value sent to server).....	954
DDEWRITE (provide data to DDE client).....	955
Object Linking and Embedding.....	957
OLE Overview.....	957
OLE Container Properties.....	959
OLEDIRECTORY (get list of installed OLE/OCX).....	968
OLE (.OCX) Custom Controls.....	970
OLE custom control Overview.....	970
.OCX Control Properties.....	971
Callback Functions.....	973
Calling OLE Object Methods.....	978
Method Syntax Overview.....	978
Parameter Passing to OLE/OCX Methods.....	981
OCX Library Procedures.....	984
OCXREGISTERPROPEDIT (install property edit callback).....	984
OCXREGISTERPROPCHANGE (install property change callback).....	985
OCXREGISTEREVENTPROC (install event processing callback).....	986
OCXUNREGISTERPROPEDIT (un-install property edit callback).....	987
OCXUNREGISTERPROPCHANGE (un-install prop change callback).....	988
OCXUNREGISTEREVENTPROC (un-install event process callback).....	989
OCXGETPARAMCOUNT (return number of parameters for current event).....	990
OCXGETPARAM (return current event parameter string).....	991
OCXSETPARAM (set current event parameter string).....	992
OCXLOADIMAGE (return an image object).....	993

Appendix B - Events **995**

Events.....	995
Field-Independent Events.....	996
EVENT:AlertKey.....	997
EVENT:BuildDone.....	997
EVENT:BuildFile.....	997
EVENT:BuildKey.....	997
EVENT:CloseDown.....	997
EVENT:CloseWindow.....	997
EVENT:Completed.....	997
DDE Events.....	998
EVENT:Docked.....	998
EVENT:Undocked.....	998
EVENT:GainFocus.....	999
EVENT:Iconize.....	999
EVENT:Iconized.....	999
EVENT:LoseFocus.....	999
EVENT:Maximize.....	999
EVENT:Maximized.....	999
EVENT:Move.....	999
EVENT:Moved.....	999
EVENT:Notify.....	1000

EVENT:OpenWindow	1000
EVENT:PreAlertKey	1000
EVENT:Restore	1000
EVENT:Restored	1000
EVENT:Resume	1000
EVENT:Size	1000
EVENT:Sized	1000
EVENT:Suspend	1001
EVENT:Timer	1001
Field-Specific Events	1002
Field-Specific Events	1002
EVENT:Accepted	1003
EVENT:AlertKey	1003
EVENT:ColumnResize	1003
EVENT:Contracted	1003
EVENT:Contracting	1003
EVENT:Drag	1003
EVENT:Dragging	1003
EVENT:Drop	1003
EVENT:DroppedDown	1004
EVENT:DroppingDown	1004
EVENT:Expanded	1004
EVENT:Expanding	1004
EVENT:Locate	1004
Mouse Events	1005
EVENT:NewSelection	1006
EVENT:PageDown	1006
EVENT:PageUp	1006
EVENT:PreAlertKey (Field Specific)	1006
EVENT:Rejected	1006
EVENT:ScrollBottom	1006
EVENT:ScrollDown	1006
EVENT:ScrollDrag	1007
EVENT:ScrollTop	1007
EVENT:ScrollTrack	1007
EVENT:ScrollUp	1007
EVENT:Selected	1007
EVENT:TabChanging	1007
Modal Events	1008

Appendix C - Runtime Properties 1009

PROP:AcceptAll	1009
PROP:Active	1010
PROP:AlwaysDrop	1010
PROP:AppInstance	1011
PROP:AssertHook	1011
PROP:AssertHook2	1012

PROP:AutoPaper	1013
PROP:BreakVar	1013
PROP:Buffer	1014
PROP:Checked	1014
PROP:Child and PROP:ChildIndex	1015
PROP:ChoiceFeg	1016
PROP:ClientHandle	1016
PROP:ClientWndProc	1017
PROP:ClipBits	1018
PROP:ColorDialogHook	1019
PROP:DDEMode	1019
PROP:DDETimeOut	1020
PROP:DeferMove	1020
PROP:Edit	1022
PROP:Enabled	1024
PROP:EventsWaiting	1026
PROP:ExeVersion	1026
PROP:FatalErrorHook	1026
PROP:FileDialogHook	1026
PROP:FlushPageNumFunc	1027
PROP:FlushPreview	1028
PROP:Follows	1029
PROP:FontDialogHook	1029
PROP:HaltHook	1030
PROP:Handle	1030
PROP:HeaderHeight	1031
PROP:HscrollPos	1032
PROP:IconList	1033
PROP:ImageBits	1035
PROP:ImageBlob	1036
PROP:InitAStringHook	1036
PROP:Interface	1036
PROP:InToolbar	1036
PROP:Items	1037
PROP:LazyDisplay	1037
PROP:LFNSupport	1038
PROP:LibHook	1038
PROP:LibVersion	1039
PROP:Line and PROP:LineCount	1039
PROP:LineHeight	1040
PROP:MaxHeight	1040
PROP:MaxWidth	1041
PROP:MessageHook	1041
PROP:MinHeight	1042
PROP:MinWidth	1042
PROP:NextField	1043
PROP:NextPageNo	1044

PROP:NextTabStop and PROP:PrevTabStop	1044
PROP:NoHeight and PROP:NoWidth	1045
PROP:NoTips	1046
PROP:NumTabs	1047
PROP:Parent	1048
PROP:Pixels	1048
PROP:PrinterDialogHook	1049
PROP:PrintMode	1049
PROP:Progress	1050
PROP:PropVscroll	1051
PROP:RejectCode	1052
PROP:ScreenText	1053
PROP:SelStart (or PROP:Selected) and PROP:SelEnd	1054
PROP:Size	1055
PROP:SnapHeight and PROP:SnapWidth	1056
PROP:StatusFont	1058
PROP:StopHook	1058
PROP:SystemPropHook	1058
PROP:TabRows	1059
PROP:Target	1060
PROP:TemplImage	1060
PROP:TemplImagePath	1060
PROP:TemplImageStatus	1060
PROP:TempNameFunc	1060
PROP:TempPagePath	1062
PROP:TempPath	1062
PROP:Thread	1062
PROP:Threading	1062
PROP:TipDelay and PROP:TipDisplay	1063
PROP:TipsFont	1063
PROP:Touched	1064
PROP:Type	1066
PROP:UpsideDown	1067
PROP:UseAddress	1068
PROP:Visible	1069
PROP:VLBproc and PROP:VLBval	1071
PROP:Vscroll	1074
PROP:VscrollPos	1075
PROP:Watched	1077
PROP:WheelScroll	1077
PROP:WindowsVersion	1077
PROP:WndProc	1078
PROP:Xorigin and PROP:YOrigin	1080
Runtime VIEW and FILE Properties	1081
PROP:Completed	1081
PROP:ConnectString	1081
PROP:CurrentKey	1082

PROP:DataPath	1082
PROP:Details	1083
PROP:DriverLogoutAlias	1083
PROP:FetchSize	1083
PROP:Field.....	1084
PROP:FieldsFile	1084
PROP:File.....	1085
PROP:Files	1085
PROP:GlobalHelp.....	1086
PROP:Held	1086
PROP:Log	1087
PROP:Logout	1088
PROP:LogoutIsolationLevel	1089
PROP:MaxStatements	1089
PROP:Profile	1089
PROP:ProgressEvents	1090
PROP:Record.....	1093
PROP:SQLDriver.....	1093
PROP:StmtAttr	1093
PROP:SupportsOp	1094
PROP:SupportsType	1094
PROP:Text (MEMO Property)	1094
PROP:Value	1095
Appendix D - Error Codes	1097
Trappable Run Time Errors.....	1097
Non-Trappable Run Time Errors	1103
Compiler Errors	1106
Specific Errors	1107
Unknown errors	1124
Appendix E - Legacy Statements	1127
Legacy Statements	1127
BOF (return beginning of file).....	1128
EOF (return end of file)	1129
FUNCTION (define a function).....	1130
POINTER (return relative record position).....	1131
SHARE (open data file for shared access).....	1132
Index:	1133

1 - Introduction

Introduction--The Language Reference Manual

Clarion is an integrated environment for writing data processing applications and management information systems for microcomputers using the Windows operating environment. Clarion's programming language is the foundation of this environment. In this manual, the language is concisely documented in a modular fashion. Although this is not a text book, you should consult this manual first when you want to know the precise syntax required to implement any declaration, statement, or function.

Wherever possible, we provide real-world example code for each item.

The Language Reference Manual

Language Reference -- Chapter Organization

CHAPTER 1 - Introduction provides an introduction to the Clarion Language Reference. It provides a brief overview of the contents of each chapter, and a guide to help the reader understand the documentation conventions used throughout the book.

CHAPTER 2 - Program Source Code Format provides the general layout of a Clarion Windows program. Punctuation, special characters, reserved words, and a detailed description of the "building blocks" required to create modular, structured Clarion source code are documented here.

CHAPTER 3 - Variable Declarations describes all the simple data types used to declare variables in a Clarion program. In addition, data display formatting masks, called "picture tokens," are defined and illustrated.

CHAPTER 4 - Entity Declarations describes all the complex data types used to declare GROUP, CLASS, FILE, VIEW, and QUEUE structures in a Clarion program.

CHAPTER 5 - Declaration Attributes describes all the attributes which can modify variable and entity declarations.

CHAPTER 6 - Windows describes the APPLICATION and WINDOW data structures and all their component structures.

CHAPTER 7 - Reports describes the REPORT data structure and all its component structures.

CHAPTER 8 - Controls describes all the controls that may be placed into APPLICATION, WINDOW, and REPORT data structures.

CHAPTER 9 - Window and Report Attributes describes all the attributes which can modify APPLICATION, WINDOW, and REPORT data structures and the controls they contain.

CHAPTER 10 - Expressions defines the syntax required to combine variables, procedures, and constants into numeric, string, or logical expressions.

CHAPTER 11 - Assignments defines all the methods to assign the value of an expression to variables. This chapter also discusses BCD operations and Clarion's Automatic Data Type Conversion.

CHAPTER 12 - Execution Control describes the compound executable statements that control program flow and operation.

CHAPTER 13 - Built-in Procedures documents all of the built-in Clarion library procedures.

APPENDIX A - DDE, OLE, and OCX documents the procedures that perform Dynamic Data Exchange (DDE), Object Linking and Embedding (OLE), and OLE Custom Controls (OCX).

APPENDIX B - Event Equates documents the EQUATE statements for events that help make Clarion code readable.

APPENDIX C - Runtime Property Assignments documents all the runtime properties.

APPENDIX D - Error Codes documents the runtime and compiler errors.

APPENDIX E - Legacy Statements documents language statements which are maintained only for backward compatibility with previous versions of Clarion.

Documentation Conventions and Symbols

Symbols are used in the syntax diagrams as follows:

<u>Symbol</u>	<u>Meaning</u>
[]	Brackets enclose an optional (not required) attribute or parameter.
()	Parentheses enclose a parameter list.
	Vertical lines enclose parameter lists, where one, but only one, of the parameters is allowed.

Coding example conventions used throughout this manual:

```

IF NOT SomeDate           !IF and NOT are keywords
  SomeDate = TODAY()    !SomeDate is a data name
END                       !TODAY and END are keywords

```

CLARION LANGUAGE KEYWORDS Any word in "All Caps" is a Clarion Language keyword

DataNames Use mixed case with caps for readability

Comments Predominantly lower case

The purpose of these conventions is to make the code examples readable and clear.

Reference Item Format

Each Clarion programming language element referenced in this manual is printed in UPPER CASE letters. Components of the language are documented with a syntax diagram, a detailed description, and source code examples.

Items are documented in logical groupings, dependent upon their hierarchical relationships. Therefore, the table of contents for this book is not listed in alphabetical order. In general, data types and structures occur at the beginning of a chapter, followed by their attributes, and executable statements and functions at the end.

The documentation format used in this book is illustrated in the syntax diagram on the following page.

KEYWORD (short description of intended use)

```
[label]  KEYWORD(  | parameter1  | [ parameter2 ] ) [ATTRIBUTE1( )] [ATTRIBUTE2( )]
          | alternate      |
          | parameter     |
          | list          |
```

KEYWORD A brief statement of what the KEYWORD does.

parameter1 A complete description of parameter1, along with how it relates to parameter2 and the KEYWORD.

alternate parameter list

A complete description of mutually exclusive alternates to parameter1, along with how they relate to parameter2 and the KEYWORD.

parameter2 A complete description of parameter2, along with how it relates to parameter1 and the KEYWORD. Because it is enclosed in brackets, [], it is optional, and may be omitted.

ATTRIBUTE1 A sentence describing the relation of ATTRIBUTE1 to the KEYWORD.

ATTRIBUTE2 A sentence describing the relation of ATTRIBUTE2 to the KEYWORD.

A concise description of what the **KEYWORD** does. In many cases the KEYWORD will be an attribute of a keyword that was described in the preceding text. Sometimes a KEYWORD has no parameters and/or attributes.

Events Generated: If the KEYWORD generates events, they are listed here.

Return Data Type: The data type returned if the KEYWORD returns a value.

Errors Posted: If KEYWORD posts errors which may be trapped by the ERROR and ERRORCODE functions, they are listed here.

Related Procedures: If KEYWORD defines a data structure, the procedures which operate on that data structure are listed here.

Example:

```
FieldOne = FieldTwo + FieldThree      !This is a source code example
FieldThree = KEYWORD(FieldOne,FieldTwo) !Comments follow the "!" character
```

See Also: Other pertinent keywords and topics

Clarion Conventions

Standard Date

A Clarion standard date is the number of days that have elapsed since December 28, 1800. The range of accessible dates is from January 1, 1801 (standard date 4) to December 31, 9999 (standard date 2,994,626). Date procedures will not return correct values outside the limits of this range. The standard date calendar also adjusts for each leap year within the range of accessible dates. Dividing a standard date by modulo 7 gives you the day of the week: zero = Sunday, one = Monday, etc.

The LONG data type with a date format (@D) display picture is normally used for a standard date. Data entry into any date format picture with a two-digit year defaults to the century of next 20 or previous 80 years. For example, entering 01/01/01 results in 01/01/2001 if the current year (per the system clock) is greater than 1980, and 01/01/1901 if the current year is 1980 or earlier.

The DATE data type is a data format used in the Btrieve Record Manager and some other file systems. A DATE field is internally converted to LONG containing the Clarion standard date before any mathematical or date procedure operation is performed. Therefore, DATE should be used for external file compatibility, and LONG is normally used for other dates.

See Also:

Date Pictures

DAY

MONTH

YEAR

TODAY

SETTODAY

DATE

Standard Time

A Clarion standard time is the number of hundredths of a second that have elapsed since midnight, plus one (1). The valid range is from 1 (defined as midnight) to 8,640,000 (defined as 11:59:59.99 PM). A standard time of one is exactly equal to midnight to allow a zero value to be used to detect no time entered. Although time is expressed to the nearest hundredth of a second, the system clock is only updated 18.2 times a second (approximately every 5.5 hundredths of a second).

The LONG data type with a time format (@T) display picture is normally used for a standard time. The TIME data type is a data format used in the Btrieve Record Manager. A TIME field is internally converted to LONG containing the Clarion standard time before any mathematical or time procedure operation is performed. Therefore, TIME should be used for external Btrieve file compatibility, and LONG should normally be used for other times.

See Also:

Time Pictures

CLOCK

SETCLOCK

Clarion Keycodes

Windows Keycode Mapping Format

Each key on the keyboard is assigned a keycode. Keycodes are 16-bit values where the low-order 8 bits (values from 0 to 255) represent the key that was pressed, and the high-order 8 bits indicate the state of the Shift, Ctrl, and Alt keys. Keycodes are returned by the KEYCODE() and KEYBOARD() procedures, and use the following format:

```

      | A | C | S | CODE |
Bits: 10  9  8 7      0

```

CODE - The Key pressed

A - Alt key bit

C - Ctrl key bit

S - Shift key bit

Calculating a keycode's numeric value is generally unnecessary, since most of the possible key combinations are listed as EQUATE statements in KEYCODES.CLW (INCLUDE this file and use the equates instead of the numbers for better code readability).

KEYCODES.CLW

Keycode equate labels assign mnemonic labels to Clarion keycodes. The keycode equates file (KEYCODES.CLW) is a Clarion source file which contains an EQUATE statement for each keycode. This file is located in the install \LIBSRC directory.

It may be merged into a source PROGRAM by placing the following statement in the global data section:

```
INCLUDE( 'KEYCODES.CLW' )
```

This file contains EQUATE statements for most of the keycodes supported by Windows. These keycode EQUATEs are used for greater code readability wherever you need to set or compare keyboard input.

See Also:

KEYCODE	KEYBOARD	
KEYCHAR	KEYSTATE	
SETKEYCODE	ALERT	ALRT

2 - Program Source Code Format

Statement Format

Clarion is a "statement oriented" language. A statement-oriented language makes use of the fact that its source code is contained in ASCII text files so every line of code is a separate record in the file. Therefore, the Carriage Return/Line Feed record delimiter can be used to eliminate punctuation.

In general, the Clarion statement format is:

```
label STATEMENT[(parameters)] [,ATTRIBUTE[(parameters)]] ...
```

Attributes specify the properties of the item and are only used on data declarations. Executable statements take the form of a standard procedure call, except assignment statements (A = B) and control structures (such as IF, CASE, and LOOP).

A statement's label must begin in column one (1) of the source code. A statement without a label must not start in column one. A statement is terminated by the end of the line. A statement too long to fit on one line can be continued by a vertical bar (|). The semi-colon is an optional statement separator that allows you to place more than one statement on a line.

Being a statement oriented language eliminates from Clarion much of the punctuation required in other languages to identify labels and separate statements. Blocks of statements are initiated by a single compound statement, and are terminated by an END statement (or period).

See Also:

Declaration and Statement Labels

Structure Termination

Field Qualification

Reserved Words

Special Characters

Declaration and Statement Labels

The language statements in a source module can be divided into two general categories: data declarations and executable statements, or simply "data" and "code."

During program execution, data declarations reserve memory storage areas that are manipulated by executable statements. A label is required for the data to be referenced in executable code. All variables, data structures, PROCEDURES, and ROUTINES are referenced by labels.

A label defines a specific location in a PROGRAM. Any code statement may be identified and referenced by a label. This allows it to be used as the target of a GOTO statement. Each label on an executable statement adds ten bytes to the executable code size, even if not referenced.

The label on a PROCEDURE statement is the procedure's name. Using the label of a PROCEDURE in an executable statement executes the procedure, or in expressions, or parameter lists of other procedures, assigns the value returned by the procedure.

The rules for valid Clarion labels are:

- A label MUST begin in column one (1) of the source code.
- A label may contain letters (upper or lower case), numerals 0 through 9, the underscore character (_), and colon (:).
- The first character must be a letter or the underscore character.
- Labels are not case sensitive (i.e. CurRent and CURRENT are the same).
- A label may not be a reserved word.
- A period (.) is valid in a label for procedures defined in a CLASS structure in the form *ClassLabel.MethodLabel*

Structure Termination

Compound data structures are created when data declarations are nested within other data declarations. There are many compound data structures within the Clarion language: APPLICATION, WINDOW, REPORT, FILE, RECORD, GROUP, VIEW, QUEUE, etc. These compound data structures must be terminated by a period (.) or the keyword END. IF, CASE, EXECUTE, LOOP, BEGIN, and ACCEPT are all executable control structures. They must also be terminated with a period or the END statement (a LOOP may optionally terminate with a WHILE or UNTIL statement).

Field Qualification

Variables declared as members of complex data structures (GROUP, QUEUE, FILE, RECORD, etc.) may have duplicate labels, as long as the duplicates are not contained within the same structure. To explicitly reference fields with duplicate labels in separate structures, you may use the PRE attribute on the structures just as it is documented (Prefix:FieldLabel) to provide unique names for each field. However, the PRE attribute is not required for this purpose and may be omitted.

Any member of any complex structure can be explicitly referenced by prepending the label of the structure containing the field to the field label, separated by a period (StructureName.FieldLabel). You must use this Field Qualification syntax to reference any field in a complex structure that does not have a PRE attribute. You may use a colon (:) instead of a period (StructureName:FieldLabel) to reference member variables of any structure except CLASS and named reference variables (this syntax is only to provide backward compatibility with previous versions of Clarion for Windows).

If the variable is within nested complex data structures, you must prepend each successive level's structure label to the variable label to explicitly reference the variable (if the nested structure has a label). If any nested structure does not have a label, then that part is omitted from the qualification sequence. This is similar to anonymous unions in C++. This means that, in the case of a GROUP structure (without a PRE attribute) in which a nested GROUP structure has a label, the fields in the inner GROUP must be referenced as *OuterGroupLabel.InnerGroupLabel.FieldLabel*. If the inner GROUP structure does not have a label, the individual fields are referenced as *OuterGroupLabel.FieldLabel*. There is one exception to this rule: the label of a RECORD structure within a FILE may be omitted so that you can reference individual fields within the file as *FileLabel.FieldLabel* instead of *FileLabel.RecordLabel.FieldLabel*.

This Field Qualification syntax is also used to reference all members of CLASS structures--both data members and methods. To call a member method of a CLASS structure, you specify *ClassName.MethodLabel* wherever the call to the PROCEDURE is valid.

To reference an element of a GROUP structure with the DIM attribute, you must specify the array element number in the Field Qualification syntax at the exact level at which the DIM attribute appears.

Example:

```

MasterFile FILE,DRIVER('TopSpeed')
Record      RECORD
AcctNumber  LONG           !Reference as Masterfile.AcctNumber
            END
            END

Detail      FILE,DRIVER('TopSpeed')
            RECORD
AcctNumber  LONG           !Reference as Detail.AcctNumber
            END
            END

Memory      GROUP,PRE(Mem)
Message     STRING(30)     !May reference as Mem:Message or Memory.Message
            END

SaveQueue   QUEUE
Field1      LONG           !Reference as SaveQueue.Field1
Field2      STRING        !Reference as SaveQueue.Field2
            END

OuterGroup  GROUP
Field1      LONG           !Reference as OuterGroup.Field1
Field2      STRING        !Reference as OuterGroup.Field2
InnerGroup  GROUP
Field1      LONG           !Reference as OuterGroup.InnerGroup.Field1
Field2      STRING        !Reference as OuterGroup.InnerGroup.Field2
            END
            END

OuterGroup  GROUP,DIM(5)
Field1      LONG           !Reference as OuterGroup[1].Field1
InnerGroup  GROUP,DIM(5)  !Reference as OuterGroup[1].InnerGroup
Field1      LONG           !Reference as OuterGroup[1].InnerGroup[1].Field1
            END
            END

```

See Also:

PRE

CLASS

Reference Variables

Reserved Words

The following keywords are reserved and may not be used as labels for any purpose:

ACCEPT	AND	BEGIN	BREAK
BY	CASE	CHOOSE	COMPILE
CYCLE	DO	ELSE	ELSIF
END	EXECUTE	EXIT	FUNCTION
GOTO	IF	INCLUDE	LOOP
MEMBER	NEW	NOT	NULL
OF	OMIT	OR	OROF
PARENT	PROCEDURE	PROGRAM	RETURN
ROUTINE	SECTION	SELF	THEN
TIMES	TO	UNTIL	WHILE
XOR			

The following keywords may be used as labels of data structures or executable statements. They may not be the label of any PROCEDURE statement. They may appear as the label of a parameter in a prototype only if the data type is also listed:

APPLICATION	CLASS	CODE	DATA
DETAIL	FILE	FOOTER	FORM
GROUP	HEADER	ITEM	ITEMIZE
JOIN	MAP	MENU	MENUBAR
MODULE	OLECONTROL	OPTION	QUEUE
RECORD	REPORT	ROW	SHEET
TAB	TABLE	TOOLBAR	VIEW
WINDOW			

Special Characters

Initiators:	!	Exclamation point begins a source code comment.
	?	Question mark begins a field equate label, label, or when used as a single character in column one of source code, designates that the statement that follows only be executed in DEBUG mode.
	@	"At" sign begins a picture token.
	*	Asterisk begins a parameter passed by address in a MAP prototype.
	~	A leading tilde on a filename indicates a file linked into the project.
Terminators:	;	Semi-colon is an executable statement separator.
	CR/LF	Carriage-return/Line-feed is an executable statement separator.
	.	Period terminates a data or code structure (a substitute for END).
		Vertical bar is the source code line continuation character.
	#	Pound sign declares an implicit LONG variable.
	\$	Dollar sign declares an implicit REAL variable.
	"	Double quote declares an implicit STRING variable.
Delimiters:	()	Parentheses enclose a parameter list.
	[]	Brackets enclose an array subscript list.
	' '	Single quotes enclose a string constant.
	{ }	Curly braces enclose a repeat count in a string constant, or a property parameter.
	< >	Angle brackets enclose an ASCII code in a string constant, or indicate a parameter in a MAP prototype which may be omitted.
	:	Colon separates the start and stop positions of a string "slice."
	,	Comma separates parameters in a parameter list.
Connectors:	.	Period is a decimal point used in numeric constants, or connects a complex structure label to the label of one of its members.
	:	Colon connects a prefix to a label.
	\$	Dollar sign connects the WINDOW or REPORT label to a field equate label in a control's property assignment expression.

Operators:	+	Plus sign indicates addition.
	-	Minus sign indicates subtraction.
	*	Asterisk indicates multiplication.
	/	Slash indicates division.
	%	Percent sign indicates modulus division.
	^	Carat indicates exponentiation.
	<	Left angle bracket indicates less than.
	>	Right angle bracket indicates greater than.
	=	Equal sign indicates assignment or equivalence.
	~	Tilde indicates the logical (Boolean) NOT operator.
	&	Ampersand indicates concatenation.
	&=	Ampersand equal indicates reference assignment or reference equivalence.
	:=	Executes a Deep Assignment statement.

Program Format

PROGRAM (declare a program)

```

PROGRAM
MAP
  prototypes
  [MODULE( )
    prototypes
  END ]
END
  global data
CODE
  statements
  [RETURN]
  procedures

```

PROGRAM	The first declaration in a Clarion program source module. Required.
MAP	Global procedure declarations. Required.
MODULE	Declare member source modules.
<i>prototypes</i>	PROCEDURE declarations.
<i>global data</i>	Declare Global data which may be referenced by all procedures.
CODE	Terminate the data declaration section and begin the executable code section of the PROGRAM.
<i>statements</i>	Executable program instructions.
RETURN	Terminate program execution. Return to operating system control .
<i>procedures</i>	Source code for the procedures in the PROGRAM module.

The **PROGRAM** statement is required to be the first declaration in a Clarion program source module. It may only be preceded by source code comments. The PROGRAM source file name is used as the object (.OBJ) and executable (.EXE) file name, when compiled. The PROGRAM statement may have a label, but the label is ignored by the compiler.

A PROGRAM with PROCEDURES must have a MAP structure. The MAP declares the PROCEDURE prototypes. Any PROCEDURE contained in a separate source file must be declared in a MODULE structure within the MAP.

Data declared in the PROGRAM module, between the keywords PROGRAM and CODE, is Global data that may be accessed by any PROCEDURE in the PROGRAM. Its memory allocation is Static.

Example:

```

PROGRAM                               !Sample program declaration
INCLUDE('EQUATES.CLW') !Include standard equates
MAP
CalcTemp  PROCEDURE                   !Procedure Prototype
END
CODE
CalcTemp                               !Call procedure
CalcTemp  PROCEDURE
Fahrenheit REAL(0)                   !Global data declarations
Centigrade REAL(0)
Window WINDOW('Temperature Conversion'),CENTER,SYSTEM
    STRING('Enter Fahrenheit Temperature: '),AT(34,50,101,10)
    ENTRY(@N-04),AT(138,49,60,12),USE(Fahrenheit)
    STRING('Centigrade Temperature:'),AT(34,71,80,10),LEFT
    ENTRY(@N-04),AT(138,70,60,12),USE(Centigrade),SKIP
    BUTTON('Another'),AT(34,92,32,16),USE(?Another)
    BUTTON('Exit'),AT(138,92,32,16),USE(?Exit)
END
CODE                                   !Begin executable code section
OPEN(Window)
ACCEPT
CASE ACCEPTED()
OF ?Fahrenheit
    Centigrade = (Fahrenheit - 32) / 1.8
    DISPLAY(?Centigrade)
OF ?Another
    Fahrenheit = 0
    Centigrade = 0
    DISPLAY
SELECT(?Fahrenheit)
OF ?Exit
    BREAK
END
END
CLOSE(Window)
RETURN

```

See Also: MAP,MODULE

PROCEDURE

Data Declarations and Memory Allocation

MEMBER (identify member source file)

```

MEMBER( [ program ] )
  [MAP
    prototypes
  END ]
[label] local data
         procedures

```

MEMBER	The first statement in a source module that is not a PROGRAM source file. Required.
<i>program</i>	A string constant containing the filename (without extension) of a PROGRAM source file. If omitted, the module is a "universal member module" that you can compile in any program by adding it to the project.
MAP	Local procedure declarations. Any procedures declared here may be referenced by the procedures in the MEMBER module.
<i>prototypes</i>	PROCEDURE declarations.
<i>local data</i>	Declare Local Static data which may be referenced only by the procedures whose source code is in the MEMBER module.
<i>procedures</i>	Source code for the procedures in the MEMBER module.

MEMBER is the first statement in a source module that is not a PROGRAM source file. It may only be preceded by source code comments. It is required at the beginning of any source file that contains PROCEDURES that are used by a PROGRAM. The MEMBER statement identifies the *program* to which the source MODULE belongs.

A MEMBER module may have a local MAP structure (which may contain MODULE structures). The procedures *prototyped* in this MAP are available for use by the other procedures in the MEMBER module. The source code for the procedures declared in this MEMBER MAP may either be contained in the MEMBER source file, or another file (if *prototyped* in a MODULE structure within the MAP).

If the *program* parameter is omitted from the MEMBER statement, you must have a MAP structure that *prototypes* the procedures it contains. You also need to INCLUDE any standard EQUATEs files that are used in your source code.

If the source code for a PROCEDURE *prototyped* in a MEMBER module's MAP is in a separate file, the *prototype* must be in a MODULE structure within the MAP. The source file MEMBER module containing the PROCEDURE definition must also contain its own MAP which declares the same *prototype* (that is, the *prototype* must appear in at least two MAP structures--the source module containing it and the source module using it). Any PROCEDURE not declared in the

Global (PROGRAM) MAP must be declared in a local MAP(s) in the MEMBER MODULE which contains its source code.

Data declared in the MEMBER module, after the keyword MEMBER and before the first PROCEDURE statement, is Member Local data that may only be accessed by PROCEDURES within the module (unless passed as a parameter). Its memory allocation is Static.

Example:

```

!Source1 module contains:
  MEMBER('OrderSys')           !Module belongs to the OrderSys program

  MAP                           !Declare local procedures
Func1 PROCEDURE(STRING),STRING !Func1 is known only in both module
  MODULE('Source2.clw')
HistOrd2 PROCEDURE             !HistOrd2 is known only in both modules
  END
  END

LocalData STRING(10)           !Declare data local to MEMBER module

HistOrd  PROCEDURE             !Declare order history procedure
HistData STRING(10)           !Declare data local to PROCEDURE
  CODE
  LocalData = Func1(HistData)

Func1 PROCEDURE(RecField)      !Declare local procedure
  CODE
  !Executable code statements

!Source2 module contains:
  MEMBER('OrderSys')           !Module belongs to the OrderSys program
  MAP                           !Declare local procedures
HistOrd2 PROCEDURE             !HistOrd2 is known only in both modules
  MODULE('Source1.clw')
Func1 PROCEDURE(STRING),STRING !Func1 is known only in both module
  END
  END
LocalData STRING(10)           !Declare data local to MEMBER module

HistOrd2 PROCEDURE             !Declare second order history procedure
  CODE
  LocalData = Func1(LocalData)

```

See Also:

MAP,MODULE,PROCEDURE,CLASS, Data Declarations and Memory Allocation

MAP (declare PROCEDURE prototypes)

```

MAP
    prototypes
    [MODULE( )
        prototypes
    END ]
END

```

MAP Contains the *prototypes* which declare the procedures and external source modules used in a PROGRAM, MEMBER module, or PROCEDURE.

prototypes Declare PROCEDURES.

MODULE Declare a member source module that contains the definitions of the *prototypes* in the MODULE.

A **MAP** structure contains the *prototypes* which declare the PROCEDURES and external source modules used in a PROGRAM, MEMBER module, or PROCEDURE which are not members of a CLASS structure.

A MAP declared in the PROGRAM source module declares *prototypes* of PROCEDURES available for use throughout the program. A MAP in a MEMBER module declares *prototypes* of PROCEDURES that are explicitly available in that MEMBER module. The same *prototypes* may be placed in multiple MEMBER modules to make them explicitly available in each. A MAP can also be included within a PROCEDURE declaration. All *prototypes* of PROCEDURES declared in a local PROCEDURE MAP may only be referenced within the PROCEDURE itself.

A MAP structure is mandatory for any non-trivial Clarion program because the BUILTINS.CLW file is automatically included in your PROGRAM's MAP structure by the compiler. This file contains *prototypes* of most of the procedures in the Clarion internal library that are available as part of the Clarion language. This file is required because the compiler does not have these *prototypes* built into it (making it more efficient). Since the *prototypes* in the BUILTINS.CLW file use some constant EQUATES that are defined in the EQUATES.CLW file, this file is also automatically included by the compiler in every Clarion program.

Example:

!One file contains:

```
PROGRAM          !Sample program in sample.cla
MAP              !Begin map declaration
LoadIt PROCEDURE ! LoadIt procedure
END             !End of map
```

!A separate file contains:

```
MEMBER('Sample') !Declare MEMBER module
MAP              !Begin MODULE local map declaration
ComputeIt PROCEDURE ! compute it procedure
END             !End of map
```

ComputeIt Procedure

LOC:Var LONG

```
MAP              ! PROCEDURE local map
  Proc1
END
```

```
Code
Proc1()
Return
```

Proc1 Procedure

```
Code
LOC:Var += 1
Return
```

See Also:

PROGRAM

MEMBER

MODULE

PROCEDURE

PROCEDURE Prototypes

MODULE (specify MEMBER source file)

```
MODULE(sourcefile)
```

```
    prototype
```

```
END
```

MODULE	Names a MEMBER module or external library file.
<i>sourcefile</i>	A string constant which contains the filename (without extension) of the Clarion language source code file containing the definitions of the PROCEDURES. If the <i>sourcefile</i> is an external library, this string may contain any unique identifier.
<i>prototype</i>	The prototype of a PROCEDURE whose definition is contained in the <i>sourcefile</i> .

A **MODULE** structure names a Clarion language MEMBER module or an external library file and contains the *prototypes* for the PROCEDURES contained in the *sourcefile*. A MODULE structure can only be declared within a MAP structure and is valid for use in any MAP structure, whether that MAP is in a PROGRAM module or MEMBER module.

Example:

```
!The "sample.clw" file contains:
PROGRAM                !Sample program in sample.clw
MAP                    !Begin map declaration
  MODULE('Loadit')    ! source module loadit.clw
LoadIt PROCEDURE      ! LoadIt procedure
  END                  ! end module
  MODULE('Compute')   ! source module compute.clw
ComputeIt PROCEDURE   ! compute it procedure
  END                  ! end module
END                    !End map
```

```
!The "loadit.clw" file contains:
MEMBER('Sample')      !Declare MEMBER module
MAP                    !Begin local map declaration
  MODULE('Process')   ! source module process.cla
ProcessIt PROCEDURE   ! process it procedure
  END                  ! end module
END                    !End map
```

See Also:

MEMBER, MAP, PROCEDURE Prototypes

PROCEDURE (define a procedure)

label **PROCEDURE** [(*parameter list*)]

local data

CODE

statements

[RETURN([*value*])]

PROCEDURE Begins a section of source code that can be executed from within a PROGRAM.

label Names the PROCEDURE. For a CLASS method's definition, this may contain the label of the CLASS prepended to the label of the PROCEDURE.

parameter list A comma delimited list of names (and, optionally, their data types) of the parameters passed to the PROCEDURE. These names define the local references within the PROCEDURE to the passed parameters. For a CLASS method's definition, this may contain the label of the CLASS (named SELF) as an implicit first parameter (if the class is not prepended to the PROCEDURE's *label*), and must always contain both the data type and parameter name.

local data Declare Local data visible only in this procedure.

CODE Terminate the data declaration section and begin the executable code section of the PROCEDURE.

statements Executable program instructions.

RETURN Terminate procedure execution. Return to the point from which the procedure was called and return the *value* to the expression in which the procedure was used (if the procedure has been prototyped to return a value).

value A numeric or string constant or variable which specifies the result of the procedure call.

PROCEDURE begins a section of source code that can be executed from within a PROGRAM. It is called by naming the PROCEDURE *label* (with its *parameter list*, if any) as an executable statement in the code section of a PROGRAM or PROCEDURE.

The *parameter list* defines the data type of each parameter (optional) followed by the label of the parameter as used within the PROCEDURE's source code (required). Each parameter is separated by a comma. The data type of each parameter (including the angle brackets if the parameter is omissible) is required along with the parameter's label if the procedure is overloaded (has multiple definitions). The *parameter list* may be exactly the same as it appears in the PROCEDURE's prototype, if that prototype contains labels for the parameters.

A PROCEDURE may contain one or more ROUTINES in its executable code *statements*. A ROUTINE is a section of executable code local to the PROCEDURE which is called with the DO statement.

A PROCEDURE terminates and returns to its caller when a RETURN statement executes. An implicit RETURN occurs at the end of the executable code. The end of executable code for the PROCEDURE is defined as the end of the source file, or the first encounter of a ROUTINE or another PROCEDURE.

A RETURN statement is required if the PROCEDURE has been prototyped to return a *value*. A PROCEDURE which has been prototyped to return a *value* can be used as an expression component, or passed as a parameter to another PROCEDURE. A PROCEDURE which has been prototyped to return a *value* may also be called in the same manner as a PROCEDURE without a RETURN *value*, if the program logic does not require the RETURN *value*. In this case, if the PROCEDURE prototype does not have the PROC attribute, the compiler will generate a warning which may be safely ignored.

Data declared within a PROCEDURE, between the keywords PROCEDURE and CODE, is Procedure Local data that can only be accessed by that PROCEDURE (unless passed as a parameter to another PROCEDURE). This data is allocated memory upon entering the procedure, and de-allocated when it terminates. If the data is smaller than the stack threshold (5K is the default) it is placed on the stack, otherwise it is allocated from the heap.

A PROCEDURE must have a prototype declared in a CLASS or the MAP of a PROGRAM or MEMBER module. If declared in the PROGRAM MAP, it is available to any other procedure in the program. If declared in a MEMBER MAP, it is available to other procedures in that MEMBER module.

Example:

```

PROGRAM                                !Example program code
MAP
OpenFile  PROCEDURE(FILE AnyFile)!Procedure prototype with parameter
ShoTime   PROCEDURE                !Procedure prototype without parameter
DayString PROCEDURE,STRING         !Procedure prototype with a return value
END

FileOne  FILE,DRIVER('Clarion')    !Declare a file
        RECORD                    !begin record declaration
Name     STRING(20)
Number   LONG
        END                      ! end record declaration
        END                      !End file declaration

TodayString STRING(9)
CODE
```

```

TodayString = DayString()      !Procedure called with a return value
OpenFile(FileOne)             !Call procedure to open file
ShoTime                       !Call ShoTime procedure
    !More executable statements

OpenFile PROCEDURE(FILE AnyFile)!Open any file
CODE                          !Begin code section
OPEN(AnyFile)                 !Open the file
IF ERRORCODE() = 2            !If file not found
    CREATE(AnyFile)           !create it
END
RETURN                        !Return to caller

ShoTime PROCEDURE             !Show time
Time LONG                    !Local variable
Window WINDOW,CENTER
    STRING(@T3),USE(Time),AT(34,70)
    BUTTON('Exit'),AT(138,92),USE(?Exit)
END
CODE                          !Begin executable code section
Time = CLOCK()                !Get time from system
OPEN(Window)
ACCEPT
CASE ACCEPTED()
    OF ?Exit
        BREAK
END
END
RETURN                        !Return to caller

DayString PROCEDURE           !Day string procedure
ReturnString STRING(9),AUTO   !Uninitialized local stack variable
CODE                          !Begin executable code section
EXECUTE (TODAY() % 7) + 1     !Find day of week from system date
    ReturnString = 'Sunday'
    ReturnString = 'Monday'
    ReturnString = 'Tuesday'
    ReturnString = 'Wednesday'
    ReturnString = 'Thursday'
    ReturnString = 'Friday'
    ReturnString = 'Saturday'
END
RETURN(ReturnString)         !Return the resulting string

```

See Also: PROCEDURE Prototypes, Data Declarations and Memory Allocation, Procedure Overloading, CLASS, ROUTINE, MAP

CODE (begin executable statements)**CODE**

The **CODE** statement separates the data declaration section from the executable statement section within a PROGRAM, PROCEDURE, or ROUTINE. The first statement executed in a PROGRAM, PROCEDURE, or ROUTINE is the statement following CODE.

Example:

```
PROGRAM

!Global Data declarations go here

CODE
!Executable statements go here

OrdList PROCEDURE           !Declare a procedure

!Local Data declarations go here

CODE                       !This is the beginning of the "code" section
!Executable statements go here
```

See Also:

PROGRAM

PROCEDURE

DATA (begin routine local data section)

DATA

The **DATA** statement begins a local data declaration section in a ROUTINE. Any ROUTINE containing a DATA section must also contain a CODE statement to terminate the data declaration section. Variables declared in a ROUTINE data section may not have the STATIC or THREAD attributes.

Example:

```
SomeProc PROCEDURE
  CODE
  !Code statements
  DO Tally                !Call the routine
  !More code statements

Tally ROUTINE           !Begin routine, end procedure
  DATA
  CountVar BYTE         !Declare local variable
  CODE
  CountVar += 1         ! increment counter
  DO CountItAgain       !Call another routine
  EXIT                  !and exit the routine
```

See Also:

CODE

ROUTINE

ROUTINE (declare local subroutine)*label* **ROUTINE** **[DATA** *local data* **CODE]** *statements*

ROUTINE Declares the beginning of a local subroutine.*label* The name of the ROUTINE. This may not duplicate the label of any PROCEDURE.**DATA** Begin data declaration statements.*local data* Declare Local data visible only in this routine.**CODE** Begin executable statements.*statements* Executable program instructions.

ROUTINE declares the beginning of a local subroutine. It is local to the PROCEDURE in which it is written and must be at the end of the CODE section of the PROCEDURE to which it belongs. All variables visible to the PROCEDURE are available in the ROUTINE. This includes all Procedure Local, Module Local, and Global data.

A ROUTINE may contain its own local data which is limited in scope to the ROUTINE in which it is declared. If local data declarations are included in the ROUTINE, they must be preceded by a DATA statement and followed by a CODE statement. Since the ROUTINE has its own name scope, the labels of these variables may duplicate variable names used in other ROUTINES or even the procedure containing the ROUTINE.

A ROUTINE is called by the DO statement followed by the label of the ROUTINE. Program control following execution of a ROUTINE is returned to the statement following the calling DO statement. A ROUTINE is terminated by the end of the source module, or by another ROUTINE or PROCEDURE. The EXIT statement can also be used to terminate execution of a ROUTINE's code (similar to RETURN in a PROCEDURE).

A ROUTINE has some efficiency issues that are not obvious:

- DO and EXIT statements are very efficient.
- Accessing procedure-level local data is less efficient than accessing module-level or global data.

- Implicit variables used only within the ROUTINE are less efficient than using local variables.
- Each RETURN statement within a ROUTINE incurs a 40-byte overhead.

Example:

```
SomeProc PROCEDURE
  CODE
  !Code statements
  DO Tally                !Call the routine
  !More code statements

Tally ROUTINE           !Begin routine, end procedure
  DATA
  CountVar BYTE         !Declare local variable
  CODE
  CountVar += 1         ! increment counter
  DO CountItAgain      !Call another routine
  EXIT                 !and exit the routine
```

See Also:

PROCEDURE

EXIT

DO

DATA

CODE

END (terminate a structure)**END**

END terminates a data declaration structure or a compound executable statement. It is functionally equivalent to a period (.).

By convention, the **END** statement is aligned in the same column as the beginning of the structure it terminates, and the code within the structure is indented for readability. **END** is usually used to terminate multi-line structures, while the period is used to terminate single-line statements. If multiple complex code structures are nested and they all terminate at the same place, multiple periods on one line are used instead of the **END** statements on multiple lines.

Example:

```
Customer FILE,DRIVER('Clarion')    !Declare a file
      RECORD                        !begin record declaration
Name      STRING(20)
Number    LONG
      END                            !end record declaration
      END                            !End file declaration

Archive FILE,DRIVER('Clarion')    !Declare a file
      RECORD                        !begin record declaration
Name      STRING(20)
Number    LONG
      END END                        !end both the record and file declarations

CODE
IF Number <> SavNumber              !Begin if structure
  DO GetNumber
END                                  !End if structure

IF SomeCondition THEN BREAK END    !Terminate with END

CASE Action                         !Begin case structure
OF 1
  DO AddRec
  IF Number <> SavNumber              !Begin if structure
    DO SomeRoutine
  END                                !End if structure
OF 2
  DO ChgRec
OF 3
  DO DelRec
END                                  !End case structure
```

Statement Execution Sequence

In the CODE section of a Clarion program, statements are normally executed line-by-line, in the sequence in which they appear in the source module. Control statements and procedure calls are used to modify this execution sequence.

PROCEDURE calls modify the execution sequence by branching to the called procedure and executing the code contained in it. Control returns to the executable statement following the procedure call when a RETURN statement is executed in the called procedure, or there are no more statements in the called procedure to execute, returning the value (if the PROCEDURE returns a value).

Control structures--IF, CASE, LOOP, ACCEPT, and EXECUTE--change the execution sequence by evaluating expressions. The control structure conditionally executes statements contained within the structure based on the evaluation of the expression(s) in the structure. ACCEPT is also a loop-type of structure, but does not evaluate any expression.

Branching also occurs with the GOTO, DO, CYCLE, BREAK, EXIT, and RETURN statements. These statements immediately and unconditionally alter the normal execution sequence.

The START procedure begins a new execution thread, unconditionally branching to that thread at the next instance of ACCEPT following the START. However, the user may choose to activate another thread by clicking the mouse on the other thread's active window.

Example:

```
PROGRAM

MAP
ComputeTime PROCEDURE(*GROUP) !Passing a group parameter
MatchMaster PROCEDURE         !Passing no parameters
END

ParmGroup GROUP                !Declare a group
FieldOne  STRING(10)
FieldTwo  LONG
END

CODE                            !Begin executable code
FieldTwo = CLOCK()              !Executes 1st
ComputeTime(ParmGroup)          !Executes 2nd, passes control to procedure
MatchMaster                     !Executes after procedure executes fully
```

PROCEDURE Calls

procname[(*parameters*)]

return = *funcname*[(*parameters*)]

<i>procname</i>	The name of the PROCEDURE as declared in the procedure's prototype.
<i>parameters</i>	An optional parameter list passed to the PROCEDURE. A parameter list may be one or more variable labels or expressions. The <i>parameters</i> are delimited by commas and are declared in the prototype.
<i>return</i>	The label of a variable to receive the value returned by the PROCEDURE.
<i>funcname</i>	The name of a PROCEDURE which returns a value, as declared in the procedure's prototype.

A PROCEDURE is called by its label (including any parameter list) as a statement in the CODE section of a PROGRAM or PROCEDURE. The parameter list must match the parameter list declared in the procedure's prototype. Procedures cannot be called in expressions.

A PROCEDURE which returns a value is called by its label (including any parameter list) as a component of an expression or parameter list passed to another PROCEDURE. The parameter list must match the parameter list declared in the procedure's prototype. A PROCEDURE which returns a value may also be called by its label (including any parameter list), in the same manner as a PROCEDURE which doesn't return a value, if its return value is not needed. This will generate a compiler warning that can be safely ignored (unless the PROC attribute is placed on its prototype).

If the PROCEDURE is a method of a CLASS, the *procname* must begin with the label of an object instance of the CLASS followed by a period then the label of the PROCEDURE (*objectname.procname*).

Example:

```

PROGRAM
MAP
ComputeTime PROCEDURE(*GROUP)      !Passing a group parameter
MatchMaster PROCEDURE,BYTE,PROC    !PROCEDURE returning a value and passing no parameter
END
ParmGroup GROUP                    !Declare a group
FieldOne  STRING(10)
FieldTwo  LONG
END
CODE
FieldTwo = CLOCK()                 !Built-in procedure called as expression
ComputeTime(ParmGroup)             !Call the compute time procedure
MatchMaster()                       !Call the procedure as a procedure

```

See Also: PROCEDURE

- REPLACE** Specify the "Construct" or "Destruct" PROCEDURE in the derived CLASS completely replaces the constructor or destructor of its parent CLASS.
- DERIVED** Specify the PROCEDURE is a derived method of a CLASS structure, There must be a matching prototype in the parent class.

All PROCEDURES in a PROGRAM must have a prototype declaration in a MAP or CLASS structure. A prototype declares to the compiler exactly what form to expect to see when the PROCEDURE is used in executable code.

There are two valid forms of prototype declarations listed in the syntax diagram on the previous page. The first one, using the PROCEDURE keyword, is valid for use everywhere and is the preferred form to use. The second form is supported only for backward compatibility with previous versions of Clarion.

A prototype contains:

- The *name* of the PROCEDURE.
- The keyword PROCEDURE is optional in a MAP structure, but required in a CLASS structure.
- An optional *parameter list* specifying all parameters that will be passed in.
- The data *return type*, if the prototype is for a PROCEDURE which will return a value.
- The parameter *calling convention*, if you are linking in objects that require stack-based parameter passing (such as objects that were not compiled with a Clarion TopSpeed compiler).
- The RAW, NAME, TYPE, DLL, PROC, PRIVATE, VIRTUAL, PROTECTED, and DERIVED attributes, as needed.

You can optionally specify the C (right to left) or PASCAL (left to right and compatible with Windows 32-bit) stack-based parameter *calling convention* for your PROCEDURE. This provides compatibility with third-party libraries written in other languages (if they were not compiled with a TopSpeed compiler). If you do not specify a *calling convention*, the default is the internal, register-based parameter passing convention used by all the TopSpeed compilers.

The RAW attribute allows you to pass just the memory address of a *?, STRING, or GROUP parameter (whether passed by value or by reference) to a non-Clarion language procedure or function. Normally, STRING or GROUP parameters pass both the address and the length of the string. The RAW attribute eliminates the length portion. This is provided for compatibility with external library functions which expect only the address of the string.

The NAME attribute provides the linker an external name for the PROCEDURE. This is also provided for compatibility with libraries written in other languages. For example: in some C language compilers, with the C calling convention specified, the compiler adds a leading underscore to the function name. The NAME attribute allows the linker to resolve the name of the function correctly.

The TYPE attribute indicates the prototype does not reference a specific PROCEDURE. Instead, it defines a prototype *name* used in other prototypes to indicate the type of procedure passed to another PROCEDURE as a parameter.

The DLL attribute specifies that the PROCEDURE prototype on which it is placed is in a .DLL. The DLL attribute is required for 32-bit applications because .DLLs are relocatable in a 32-bit flat address space, which requires one extra dereference by the compiler to address the procedure.

The PRIVATE attribute specifies that only another PROCEDURE that is in the same MODULE may call it. This would most commonly be used on a prototype in a module's MAP structure, but may also be used in the global MAP.

When the *name* of a prototype is used in the *parameter list* of another prototype, it indicates the procedure being prototyped will receive the label of a PROCEDURE that receives the same *parameter list* (and has the same *return type*, if it returns a value). A prototype with the TYPE attribute may not also have the NAME attribute.

Example:

```
MAP
  MODULE('Test')
    MyProc1 PROCEDURE(LONG)           !'test.clw' contains these procedures
    MyProc2 PROCEDURE(<*LONG>)        !LONG value-parameter
    MyProc3 PROCEDURE(LONG=23)        !Omittable LONG variable-parameter
    END                                 !Passes 23 if omitted
  MODULE('Party3.Obj')
    Func46 PROCEDURE(*CSTRING),REAL,C,RAW !A third-party library
    Func47 PROCEDURE(*CSTRING),*CSTRING,C,RAW !Pass CSTRING address-only to C function
    Func48 PROCEDURE(REAL),REAL,PASCAL !Returns pointer to a CSTRING
    Func49 PROCEDURE(SREAL),REAL,C,NAME('_func49') !PASCAL calling convention
    END                                 !C convention and external function name
  MODULE('STDFuncs.DLL')
    Func50 PROCEDURE(SREAL),REAL,PASCAL,DLL !A standard functions .DLL
    END
END
```

See Also:

MAP, MEMBER, MODULE, NAME

PROCEDURE, RETURN, Prototype Parameter Lists

Procedure Overloading, CLASS

Prototype Parameter Lists - General Syntax

	<code>[CONST] type [label]</code>
	<code><[CONST] type [label] ></code>
	<code>type [label] = default</code>
CONST	An optional qualifier for the parameter which is valid only on a variable-parameter. This means that the parameter being passed by address may not be updated in the procedure. It is treated as if it were a constant value.
<i>type</i>	The data type of the parameter. This may be a value-parameter, variable-parameter, array, unspecified data type, entity, procedure-parameter, or a named GROUP, QUEUE, or CLASS.
<i>label</i>	An optional documentary label for the parameter. This label is not required and is placed in the prototype for documentation purposes only.
<code>< ></code>	Angle brackets indicate the parameter is omissible. The OMITTED procedure detects the omission. All parameter <i>types</i> can be omitted.
<code>= default</code>	A <i>default</i> value indicates the numeric parameter is omissible, and if omitted, the <i>default</i> value is passed. The OMITTED procedure will not detect the omission--a value is passed. Valid only on simple numeric <i>types</i> .

The *parameter list* in a PROCEDURE prototype is a comma-delimited list of the data *types* to pass to the PROCEDURE. The entire *parameter list* is enclosed in the parentheses following the PROCEDURE keyword (or the *name*). Each parameter's *type* may be followed by a space then a valid Clarion *label* for the parameter (which is ignored by the compiler and only documents the purpose of the parameter). Each numeric value-parameter (passed by value) may also include an assignment of a constant value to the *type* (or the documentary *label*, if present) that defines the default value to pass if the parameter is omitted.

Any parameter that may be omitted when the PROCEDURE is called must be included in the prototype's *parameter list* and enclosed in angle brackets (`< >`) unless a *default* value is defined for the parameter. The OMITTED procedure allows you to test for unpassed parameters at runtime (except those parameters which have a *default* value).

Example:

```

MAP
  MODULE( 'Test ' )
MyProc1 PROCEDURE(LONG)                !LONG value-parameter
MyProc2 PROCEDURE(<LONG>)              !Omissible LONG value-parameter
MyProc3 PROCEDURE(LONG=23)             !Passes 23 if omitted
MyProc4 PROCEDURE(LONG Count, REAL Sum) !LONG passing a Count and REAL passing a Sum
MyProc5 PROCEDURE(LONG Count=1,REAL Sum=0)!Count defaults to 1 and Sum to 0
  END
END

```

Value-parameters

Value-parameters are "passed by value." A copy of the variable passed in the parameter list of the "calling" PROCEDURE is used in the "called" PROCEDURE. The "called" PROCEDURE cannot change the value of the variable passed to it by the "caller." Simple assignment data conversion rules apply; Value-parameters actually passed are converted to the data type in the PROCEDURE prototype. Valid value-parameters are:

BYTE SHORT USHORT LONG ULONG SREAL REAL DATE TIME STRING

Example:

```
MAP
  MODULE('Test')
MyProc1 PROCEDURE(LONG)                !LONG value-parameter
MyProc2 PROCEDURE(<LONG>)              !Omittable LONG value-parameter
MyProc3 PROCEDURE(LONG=23)            !Passes 23 if omitted
MyProc4 PROCEDURE(LONG Count, REAL Sum) !LONG passing Count and REAL passing a Sum
MyProc5 PROCEDURE(LONG Count=1, REAL Sum=0)!Count defaults to 1 and Sum to 0
  END
  MODULE('Party3.Obj')
Func48 PROCEDURE(REAL),REAL,PASCAL      !PASCAL calling convention
Func49 PROCEDURE(SREAL),REAL,C,NAME('_func49')!C convention and external function name
  END
END
```

Variable-parameters

Variable-parameters are "passed by address." A variable passed by address has only one memory address. Changing the value of the variable in the "called" PROCEDURE also changes its value in the "caller." Variable-parameters are listed by data type with a leading asterisk (*) in the PROCEDURE prototype in the MAP. Valid variable-parameters are:

*BYTE *SHORT *USHORT *LONG *ULONG *SREAL *REAL *BFLOAT4 *BFLOAT8
 *DECIMAL *PDECIMAL *DATE *TIME *STRING *PSTRING *CSTRING *GROUP

Example:

```
MAP
  MODULE('Test')
MyProc2 PROCEDURE(<*LONG>)              !Omittable LONG variable-parameter
MyFunc1 PROCEDURE(*SREAL),REAL,C      !SREAL variable-parameter, REAL return, C call
MyProc6 PROCEDURE(CONST *CSTRING Value) !Value retains a constant value in procedure
  END
  MODULE('Party3.Obj')
Func4  PROCEDURE(*CSTRING),REAL,C,RAW  !Pass CSTRING address-only to C function
Func47 PROCEDURE(*CSTRING),CSTRING,C,RAW !Returns pointer to a CSTRING
  END
END
```

Passing Arrays

To pass an entire array as a parameter, the prototype must declare the array's data type as a Variable-parameter ("passed by address") with an empty subscript list. If the array has more than one dimension, commas (as position holders) must indicate the number of dimensions in the array. The calling statement must pass the entire array to the PROCEDURE, not just one element.

Example:

```

MAP
MainProc PROCEDURE
AddCount PROCEDURE(*LONG[,] Total,*LONG[,] Current) !Passing two 2-dimensional arrays
  END
  CODE
  MainProc !Call first procedure
MainProc PROCEDURE
TotalCount LONG,DIM(10,10)
CurrentCnt LONG,DIM(10,10)
  CODE
  AddCount(TotalCount,CurrentCnt) !Call the procedure passing arrays
AddCount PROCEDURE(*LONG[,] Total,*LONG[,] Current) !Procedure expects two arrays
  CODE
  LOOP I# = 1 TO MAXIMUM(Total,1) !Loop through first subscript
    LOOP J# = 1 TO MAXIMUM(Total,2) !Loop through second subscript
      Total[I#,J#] += Current[I#,J#] !incr TotalCount from CurrentCnt
    END
  END
  CLEAR(Current) !Clear CurrentCnt array

```

Parameters of Unspecified Data Type

You can write general purpose procedures which perform operations on passed parameters where the exact data type of the parameter may vary from one call to the next by using **untyped value-parameters** and **untyped variable-parameters**. These are polymorphic parameters; they may become any other simple data type depending upon the data type passed to the procedure.

Untyped value-parameters are represented in the prototype with a question mark (?). When the procedure executes, the parameter is dynamically typed and acts as a data object of the base type (LONG, DECIMAL, STRING, or REAL) of the passed variable, or the base type of whatever it was last assigned. This means that the "assumed" data type of the parameter can change within the PROCEDURE, allowing it to be treated as any data type.

An untyped value-parameter is "passed by value" to the PROCEDURE and its assumed data type is handled by Clarion's automatic Data Conversion Rules. Data types which may be passed as untyped value-parameters:

```

BYTE SHORT USHORT LONG ULONG SREAL REAL BFLOAT4 BFLOAT8 DECIMAL
PDECIMAL DATE TIME STRING PSTRING CSTRING GROUP (treated as a
STRING) Untyped value-parameter (?) Untyped Variable-parameter (**)
```

The RAW attribute is valid for use if the untyped value-parameter (?) is being passed to external library functions written in other languages than Clarion. This converts the data to a LONG then passes the data as a C/C++ "void *" parameter (which eliminates "type inconsistency" warnings).

Untyped variable-parameters are represented in the PROCEDURE prototype with an asterisk and a question mark (*?). Within the procedure, the parameter acts as a data object of the type of the variable passed in at runtime. This means the data type of the parameter is fixed during the execution of the PROCEDURE.

An untyped variable-parameter is "passed by address" to the PROCEDURE. Therefore, any changes made to the passed parameter within the PROCEDURE are made directly to the variable which was passed in. This allows you to write polymorphic procedures.

Within a procedure which receives an untyped variable-parameter, it is not safe to make any assumptions about the data type coming in. The danger of making assumptions is the possibility of assigning an out-of-range value which the variable's actual data type cannot handle. If this happens, the result may be disastrously different from that expected.

Data types which may be passed as untyped variable-parameters:

```

BYTE SHORT USHORT LONG ULONG SREAL REAL BFLOAT4 BFLOAT8 DECIMAL
PDECIMAL DATE TIME STRING PSTRING CSTRING
Untyped variable-parameter (**)
```

The RAW attribute is valid for use if the untyped variable-parameter (*?) is being passed to external library functions written in other languages than Clarion. This has the same effect as passing a C or C++ "void *" parameter.

Arrays may not be passed as either kind of untyped parameter.

Example:

```

PROGRAM
MAP
Proc1 PROCEDURE(?)           !Untyped value-parameter
Proc2 PROCEDURE(**)         !Untyped variable-parameter
Proc3 PROCEDURE(**)         !Untyped variable-parameter (set to crash)
Max  PROCEDURE(?,?,?)      !Procedure returning Untyped value-parameter
END
GlobalVar1 BYTE(3)          !BYTE initialized to 3
GlobalVar2 DECIMAL(8,2,3)
```

```

GlobalVar3  DECIMAL(8,1,3)
MaxInteger  LONG
MaxString   STRING(255)
MaxFloat    REAL
CODE
Proc1(GlobalVar1)          !Pass in a BYTE, value is 3
Proc2(GlobalVar2)          !Pass it a DECIMAL(8,2), value is 3.00 - it prints 3.33
Proc2(GlobalVar3)          !Pass it a DECIMAL(8,1), value is 3.0 - it prints 3.3
Proc3(GlobalVar1)          !Pass it a BYTE and watch it crash
MaxInteger = Max(1,5)      !Max procedure returns the 5
MaxString = Max('Z','A') !Max procedure returns the 'Z'
MaxFloat = Max(1.3,1.25)  !Max procedure returns the 1.3

Proc1 PROCEDURE(? ValueParm)
CODE                               !ValueParm starts at 3 and is a LONG
ValueParm = ValueParm & ValueParm !Now Contains '33' and is a STRING
ValueParm = ValueParm / 10         !Now Contains 3.3 and is a REAL

Proc2 PROCEDURE(*? VariableParm)
CODE
VariableParm = 10 / 3              !Assign 3.33333333... to passed variable

Proc3 PROCEDURE(*? VariableParm)
CODE
LOOP
  IF VariableParm > 250 THEN BREAK. !If passed a BYTE, BREAK will never happen
  VariableParm += 10
END

Max PROCEDURE(Val1,Val2)           !Find the larger of two passed values
CODE
IF Val1 > Val2                     !Check first value against second
  RETURN(Val1)                     ! return first, if largest
ELSE
  RETURN(Val2)                     ! otherwise
  ! return the second
END

```

See Also:

MAP

MEMBER

MODULE

PROCEDURE

CLASS

Entity-parameters

Entity-parameters pass the name of a data structure to the "called" PROCEDURE. Passing the entity allows the "called" PROCEDURE to use those Clarion commands that require the label of the structure as a parameter. Entity-parameters are listed by entity type in the PROCEDURE prototype in the MAP. Entity-parameters are always "passed by address." Valid entity-parameters are:

```
FILE VIEW KEY INDEX QUEUE WINDOW REPORT BLOB
```

A REPORT can be passed as the parameter to a procedure prototyped to receive a WINDOW, since internally they use the same passing structure.

Example:

```
MAP
  MODULE('Test')
MyFunc2  PROCEDURE(FILE),STRING      !FILE entity-parameter, returning a STRING
ProcType PROCEDURE(FILE),TYPE        !Procedure-parameter type definition
MyFunc4  PROCEDURE(FILE),STRING,PROC !May be called as a procedure without warnings
MyProc6  PROCEDURE(FILE),PRIVATE     !May only be called by other procs in TEST.CLW
  END
END
```

Procedure-parameters

Procedure-parameters pass the name of another PROCEDURE to the "called" PROCEDURE. Procedure-parameters are listed by the name of a prototype of the same type in the PROCEDURE prototype in the MAP (which may or may not have the TYPE attribute). When called in executable code, the "called" PROCEDURE must be passed the name of a PROCEDURE whose prototype is exactly the same as the procedure named in the "called" procedure's prototype.

Each parameter in the list may be followed by a valid Clarion label which is completely ignored by the compiler. This label is used only to document the parameter to make the prototype more readable, or to duplicate the PROCEDURE definition statement. Each passed parameter's definition may also include the assignment of a constant value to the data type (or the documentary label, if present) that defines the default value to pass if the parameter is omitted.

Example:

```
MAP
  MODULE('Test')
ProcType PROCEDURE(FILE),TYPE        !Procedure-parameter type definition
MyFunc3  PROCEDURE(ProcType),STRING !ProcType procedure-parameter, returning a STRING,
  END                                     !must be passed a procedure that takes a FILE
  END                                     !as a parameter
```

Passing Named GROUPs, QUEUEs, and CLASSEs

Passing a GROUP as a Variable-Parameter, or a QUEUE as an Entity-Parameter, to a PROCEDURE does not allow you to reference the component fields within the structure in the receiving PROCEDURE . You can alternatively pass a "named" GROUP or QUEUE to achieve this. You may also name a CLASS in the same manner to allow the receiving procedure to access the public data members and methods of the CLASS.

To reference the component fields within the structure, place the label of a GROUP, QUEUE, or CLASS structure in the receiving PROCEDURE's prototype *parameter list* as the data type for the parameter. This passes the parameter "by address" and allows the receiving procedure to reference the component fields of the structure (and the public methods of a CLASS pass in this manner).

The data actually passed as the parameter must always have a similar structure (defined with the same data types) for its component fields. The GROUP or QUEUE actually passed can be a "superset" of the named parameter, as long as the first fields in the "superset" group are the same as the GROUP or QUEUE named in the prototype. The actually passed CLASS object can also be a derived class of the CLASS named in the prototype. The "extra" fields in the passed GROUP, QUEUE, or CLASS are not available for use in the receiving procedure.

The GROUP, QUEUE, or CLASS named in the *parameter list* does not need to have the TYPE attribute, and does not have to be declared before the procedure's prototype, but it must be declared before the PROCEDURE that will receive the parameter is called. This is the only instance in the Clarion language where the compiler allows such a "forward reference."

Use Field Qualification syntax to reference the members of the passed group in the receiving procedure (LocalName.MemberName). The member fields of the structure are referenced by the labels given them in the group named as the data type in the prototype--not the labels of the fields in the structure actually passed in. This allows the receiving procedure to be completely generic, regardless of what actual data structure is passed to it.

Example:

```

PROGRAM
MAP
MyProc PROCEDURE
AddQue PROCEDURE(PassGroup PassedGroup, NameQue PassedQue)
    END                !AddQue receives a GROUP defined like PassGroup and
                        ! a QUEUE defined like NameQue

PassGroup  GROUP,TYPE  !Type definition -- no memory allocated
F1         STRING(20) !  GROUP with 2 STRING(20) fields
F2         STRING(20)
END

```

```

NameGroup  GROUP          !Name group
First      STRING(20)     !first name
Last       STRING(20)     !last name
Company    STRING(30)     !This extra field is not available to the receiving
                END      !procedure (AddQue) since PassGroup only has two fields

NameQue    QUEUE,TYPE    !Name Queue, Type definition -- no memory allocate
First      STRING(20)
Last       STRING(20)
                END

CODE
MyProc

MyProc  PROCEDURE
LocalQue  NameQue        !Local Name Queue, declared exactly the same as Na

CODE
NameGroup.First = 'Fred'
NameGroup.Last = 'Flintstone'
AddQue(NameGroup,LocalQue)    !Pass NameGroup and LocalQue to AddQue procedure

NameGroup.First = 'Barney'
NameGroup.Last = 'Rubble'
AddQue(NameGroup,LocalQue)

NameGroup.First = 'George'
NameGroup.Last = 'O''Jungle'
AddQue(NameGroup,LocalQue)

LOOP X# = 1 TO RECORDS(LocalQue)    !Look at what's in the LocalQue now
  GET(LocalQue,X#)
  MESSAGE(CLIP(LocalQue.First) & ' ' & LocalQue.Last)
END

AddQue  PROCEDURE(PassGroup PassedGroup, NameQue PassedQue)
CODE
PassedQue.First = PassedGroup.F1    !Effectively: LocalQue.First = NameGroup.First
PassedQue.Last = PassedGroup.F2    !Effectively: LocalQue.Last = NameGroup.Last
ADD(PassedQue)                      !Add an entry into the PassedQue (LocalQue)
ASSERT(NOT ERRORCODE())

```

See Also:

MAP, MEMBER, MODULE, PROCEDURE, CLASS

PROCEDURE Return Types

A PROCEDURE prototyped with a return value must RETURN a value. The data type to return is listed, separated by a comma, after the optional parameter list.

Value RETURN types:

```

BYTE   SHORT   USHORT  LONG   ULONG   SREAL  REAL   DATE
TIME   STRING  Untyped value-parameter (?)

```

An untyped value-parameter return value (?) indicates the data type of the value returned by the PROCEDURE is not known. This functions in exactly the same manner as an untyped value-parameter. When the value is returned from the PROCEDURE, standard Clarion Data Conversion Rules apply, no matter what data type is returned.

Variable RETURN types:

```

CSTRING *STRING *BYTE   *SHORT *USHORT *LONG
*ULONG  *SREAL *REAL   *DATE  *TIME
Untyped variable-parameter (*?)

```

Variable return types are provided just for prototyping external library functions (written in another language) which return only the address of data--they are not valid for use on Clarion language procedures.

Functions which return pointers (the address of some data) should be prototyped with an asterisk prepended to the return data type (except CSTRING). The compiler automatically handles the returned pointer at runtime. Functions prototyped this way act just like a variable defined in the program--when the function is used in Clarion code, the data referenced by the returned pointer is automatically used. This data can be assigned to other variables, passed as parameters to procedures, or the ADDRESS function may return the address of the data.

CSTRING is an exception because all the others are fixed length datums, and a CSTRING is not. So, any C function that returns a pointer to a CSTRING can be prototyped as "char *" at the C end, but the compiler thinks the procedure and copies the datum onto the stack. Therefore, just like the other pointer return values, when the function is used in Clarion code the data referenced by the returned pointer is automatically used (the pointer is dereferenced).

As an example of this, assume that the XYZ() function returns a pointer to a CSTRING, CStringVar is a CSTRING variable, and LongVar is a LONG variable. The simple Clarion assignment statement, CStringVar = XYZ(), places the data referenced by the XYZ() function's returned pointer, in the CStringVar variable. The assignment, LongVar = ADDRESS(XYZ()), places the memory address of that data in the LongVar variable.

An untyped variable-parameter return value (*?) indicates the data type of the variable returned by the PROCEDURE is not known. This functions in exactly the same manner as an untyped variable-parameter.

Reference RETURN types:

```

*FILE *KEY *WINDOW *VIEW
Named CLASS (*ClassName)
Named QUEUE (*QueueName)

```

A PROCEDURE may return a reference which may either be assigned to a reference variable, or used in a parameter list wherever the referenced object would be appropriate. A PROCEDURE that returns *WINDOW may also return the label of an APPLICATION or REPORT structure. NULL is a valid value to return.

Example:

```

MAP
MODULE('Party3.Obj') !A third-party library
Func46 PROCEDURE(*CSTRING),REAL,C,RAW
        !Pass CSTRING address-only to C function, return REAL
Func47 PROCEDURE(*CSTRING),CSTRING,C,RAW
        !Returns pointer to a CSTRING
Func48 PROCEDURE(REAL),REAL,PASCAL
        !PASCAL calling convention, return REAL
Func49 PROCEDURE(SREAL),REAL,C,NAME('_func49')
        !C convention and external function name, return REAL

END
END

```

See Also:

MAP

MEMBER

MODULE

NAME

PROCEDURE

RETURN

Reference Variables

Prototype Attributes

C, PASCAL (parameter passing conventions)

C

PASCAL

The **C** and **PASCAL** attributes of a PROCEDURE prototype specifies that parameters are always passed on the stack. The C convention passes the parameters from right to left as they appear in the parameter list, while the PASCAL convention passes them from left to right. PASCAL is also completely compatible with the Windows API calling convention for 32-bit compiled applications-- it is the Windows-standard calling convention (and also disables name mangling).

These calling conventions provide compatibility with third-party libraries written in other languages (if they were not compiled with a TopSpeed compiler). If you do not specify a calling convention in the prototype, the default calling convention is the internal, register-based parameter passing convention used by all the TopSpeed compilers.

Example:

```
MAP
  MODULE('Party3.Obj')
Func46 PROCEDURE(*CSTRING,*REAL),REAL,C,RAW      !A third-party library
Func49 PROCEDURE(*CSTRING,*REAL),REAL,PASCAL,RAW !Pass REAL then CSTRING, address-only
  END
  END
```

See Also:

PROCEDURE Prototypes

Prototype Parameter Lists

DERIVED (prevent function overloading)

DERIVED

The **DERIVED** attribute of a PROCEDURE prototype specifies that the PROCEDURE on whose prototype it is placed is a VIRTUAL procedure. It can be used by itself or in addition to the VIRTUAL attribute on the prototype.

Use DERIVED to force a compile error when non-matching prototypes between a DERIVED virtual method and its PARENT are found. This prevents function overloading when a VIRTUAL method is the intention.

Example:

```
ClassA CLASS
Method1 PROCEDURE(LONG, <LONG>)
    END
ClassB CLASS(ClassA)
Method2 PROCEDURE(LONG,<LONG>),DERIVED
    END
ClassC CLASS(ClassA)
Method3 PROCEDURE(LONG,<LONG>),VIRTUAL,DERIVED
    END
ClassD CLASS(ClassA)
Method4 PROCEDURE(STRING),DERIVED !Will produce compiler error
    END
```

See Also:

CLASS

VIRTUAL

DLL (set procedure defined externally in .DLL)

DLL([*flag*])

DLL	Declares a PROCEDURE defined externally in a .DLL.
<i>flag</i>	A numeric constant, equate, or Project system define which specifies the attribute as active or not. If the <i>flag</i> is zero, the attribute is not active, just as if it were not present. If the <i>flag</i> is any value other than zero, the attribute is active. Uniquely, it may be an undefined label, in which case the attribute is active.

The **DLL** attribute specifies that the PROCEDURE on whose prototype it is placed is defined in a .DLL. The DLL attribute is required for 32-bit applications because .DLLs are relocatable in a 32-bit flat address space, which requires one extra dereference by the compiler to address the procedure.

Example:

```
MAP
  MODULE('STDFuncs.DLL')           !A standard functions .DLL
Func50 PROCEDURE(SREAL),REAL,PASCAL,DLL(dll_mode) !
  END
END
```

See Also:

EXTERNAL

NAME (set prototype's external name)

NAME(*constant*)

NAME Specifies an "external" name for the linker.

constant A string constant containing the external name to assign. This is case sensitive.

The **NAME** attribute specifies an "external" name for the linker. The **NAME** attribute may be placed on a PROCEDURE Prototype. The *constant* supplies the external name used by the linker to identify the procedure or function from an external library or to provide a Clarion language prototype with an external name for external linkage (usually to eliminate the compiler's standard name mangling), making it easier to construct an export list for a .DLL to be used in other language projects.

Example:

```
PROGRAM
MAP
MODULE('External.Obj')
AddCount PROCEDURE(LONG),LONG,C,NAME('_AddCount') !C function named '_AddCount'
END
END
```

See Also:

PROCEDURE Prototypes

Name Mangling and C++ Compatibility

PRIVATE (set procedure private to a CLASS or module)**PRIVATE**

The **PRIVATE** attribute specifies that the PROCEDURE on whose prototype it is placed may be called only from another PROCEDURE within the same source MODULE. This encapsulates it from other modules.

PRIVATE is normally used on method prototypes in CLASS structures, so that the method may only be called from the other CLASS methods in the module. PRIVATE methods are not inherited by CLASSES derived from the CLASS containing the PRIVATE method's prototype, although they can be VIRTUAL if the derived CLASS is contained in the same module.

Example:

```
MAP
  MODULE('STDFuncs.DLL')           !A standard functions .DLL
Func49 PROCEDURE(SREAL),REAL,PASCAL,PROC
Proc50 PROCEDURE(SREAL),PRIVATE   !Callable only from Func49
  END
END
```

```
OneClass CLASS,MODULE('OneClass.CLW'),TYPE
BaseProc PROCEDURE(REAL Parm)     !Public method
Proc    PROCEDURE(REAL Parm),PRIVATE !Declare a private method
  END
```

```
TwoClass OneClass                !Instance of OneClass
```

```
CODE
TwoClass.BaseProc(1)             !Legal call to BaseProc
TwoClass.Proc(2)                 !Illegal call to Proc
```

```
!In OneClass.CLW:
MEMBER()
```

```
OneClass.BaseProc PROCEDURE(REAL Parm)
CODE
SELF.Proc(Parm)                 !Legal call to Proc
```

```
OneClass.Proc    PROCEDURE(REAL Parm)
CODE
RETURN(Parm)
```

See Also: CLASS

PROC (set function called as procedure without warnings)

PROC

The **PROC** attribute may be placed on a PROCEDURE prototyped with a return value. This allows you to use it as normal a PROCEDURE call, not only in expressions and assignments, for those instances in which you do not need the return value. The PROC attribute suppresses the compiler warnings you would otherwise get from such use.

Example:

```
MAP
  MODULE('STDFuncs.DLL')    !A standard functions .DLL
Func50 PROCEDURE(SREAL),REAL,PASCAL,PROC
  END
END
```

See Also:

PROCEDURE

PROTECTED (set procedure private to a CLASS or derived CLASS)**PROTECTED**

The **PROTECTED** attribute specifies that the PROCEDURE on whose prototype it is placed is visible only to the PROCEDURES declared within the same CLASS structure (the other methods of that CLASS) and the methods of any CLASS derived from the CLASS. This encapsulates the PROCEDURE from being called from any code external to the CLASS within which it is prototyped or subsequently derived CLASSES.

Example:

```

OneClass CLASS,MODULE( 'OneClass.CLW' ),TYPE
BaseProc  PROCEDURE(REAL Parm)                !Public method
Proc      PROCEDURE(REAL Parm),PROTECTED      !Declare a protected method
        END

TwoClass OneClass                             !Instance of OneClass

ThreeClass CLASS(OneClass),MODULE( 'ThreeClass.CLW' ) !Derived from OneClass
ThreeProc  PROCEDURE(REAL Parm)                !Declare a Public method
        END

CODE
TwoClass.BaseProc(1)                          !Legal call to BaseProc
TwoClass.Proc(2)                              !Illegal call to Proc

!In OneClass.CLW:
MEMBER( )

OneClass.BaseProc PROCEDURE(REAL Parm)
CODE
SELF.Proc(Parm)                               !Legal call to Proc

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

!In ThreeClass.CLW:
MEMBER( )

ThreeClass.NewProc PROCEDURE(REAL Parm)
CODE
SELF.Proc(Parm)                               !Legal call to Proc

```

See Also: CLASS

RAW (pass address only)

RAW

The **RAW** attribute of a PROCEDURE prototype specifies that STRING or GROUP parameters pass the memory address only. This allows you to pass just the memory address of a *?, STRING, or GROUP parameter, whether passed by value or by reference, to a non-Clarion language procedure or function. Normally, STRING or GROUP parameters pass the address and the length of the string. The RAW attribute eliminates the length portion. For a prototype with a ? parameter, the parameter is taken as a LONG but passed as a "void *" which just eliminates linker warnings. This is provided for compatibility with external library functions that expect only the address of the string.

If a function is prototyped with one of the following return types: ?, *? or *STRING, and the prototype has the RAW attribute, the return value is treated as a LONG.

Example:

```
MAP
  MODULE('Party3.Obj')                !A third-party library
Func46 PROCEDURE(*CSTRING),REAL,C,RAW !Pass CSTRING address-only to C function
  END
END
```

See Also:

PROCEDURE Prototypes

Prototype Parameter Lists

REPLACE (set replacement constructor or destructor)**REPLACE**

The **REPLACE** attribute specifies that the PROCEDURE on whose prototype it is placed completely replaces the constructor or destructor from its parent class. REPLACE is valid only on a PROCEDURE labelled either "Construct" or "Destruct" and declared within a CLASS structure which is derived from a class which also contains a matching "Construct" or "Destruct" PROCEDURE. If the PROCEDURE label is "Construct" the method is a Constructor--automatically called when the object is instantiated. An object is instantiated when it comes into scope or when created with a NEW statement. If the PROCEDURE label is "Destruct" the method is a Destructor--automatically called when the object is destroyed. An object is destroyed when it goes out of scope or when destroyed with a DISPOSE statement.

Example:

```

PROGRAM
SomeQueue QUEUE,TYPE
F1      STRING(10)
      END
OneClass CLASS,MODULE('OneClass.CLW'),TYPE
ObjectQueue &SomeQueue      !Declare a reference to a named queue
Construct  PROCEDURE        !Declare a Constructor
Destruct   PROCEDURE        !Declare a Destructor
      END
TwoClass CLASS(OneClass),MODULE('TwoClass.CLW'),TYPE
Construct  PROCEDURE,REPLACE !Declare a replacement Constructor
      END
MyClass   OneClass          !Instance of OneClass
YourClass &TwoClass        !Reference to TwoClass
CODE      !MyClass object comes into scope,
          !autocalling OneClass.Construct

YourClass &= NEW(TwoClass) !YourClass object comes into scope,
          !autocalling TwoClass.Construct

DISPOSE(YourClass)        !YourClass object goes out of scope,
          !autocalling OneClass.Destruct

RETURN                    !MyClass object goes out of scope,
          !autocalling OneClass.Destruct

!OneClass.CLW contains:
OneClass.Construct PROCEDURE
CODE
  SELF.ObjectQueue = NEW(SomeQueue) !Create the object's queue
OneClass.Destruct  PROCEDURE
CODE
  FREE(SELF.ObjectQueue)           !Free the queue entries
  DISPOSE(SELF.ObjectQueue)        ! and remove the queue
!TwoClass.CLW contains:
TwoClass.Construct PROCEDURE

```

CODE

```
SELF.ObjectQueue = NEW(SomeQueue) !Create the object's queue
SELF.ObjectQueue.F1 = 'First Entry'
ADD(SELF.ObjectQueue)
```

See Also:

NEW, DISPOSE, CLASS

TYPE (specify PROCEDURE type definition)

TYPE

The **TYPE** attribute specifies a prototype that does not reference an actual PROCEDURE. Instead, it defines a prototype *name* to use in other prototypes to indicate the type of procedure passed to another PROCEDURE as a parameter.

When the *name* of the TYPEd prototype is used in the *parameter list* of another prototype, the procedure being prototyped will receive, as a passed parameter, the label of a PROCEDURE that has the same type of *parameter list* (and has the same *return type*, if it returns a value).

Example:

```
MAP
ProcType PROCEDURE(FILE),TYPE      !Procedure-parameter type definition
MyFunc3  PROCEDURE(ProcType),STRING !ProcType procedure-parameter, returning a STRING,
END                                     ! must be passed the label of a procedure that
                                       !takes a FILE as a required parameter
```

See Also:

PROCEDURE Prototypes

Prototype Parameter Lists

VIRTUAL (set virtual method)**VIRTUAL**

The **VIRTUAL** attribute specifies that the PROCEDURE on whose prototype it is placed is a virtual method of the CLASS containing the prototype. This allows methods in a parent CLASS to access methods in a derived CLASS. The VIRTUAL attribute must be placed on both the method's parent class prototype and the derived class's prototype.

Example:

```

OneClass CLASS                                !Base class
BaseProc  PROCEDURE(REAL Parm)              !Non-virtual method
Proc      PROCEDURE(REAL Parm),VIRTUAL !Declare a virtual method
        END

TwoClass CLASS(OneClass)                    !Derived class of OneClass
Proc      PROCEDURE(REAL Parm),VIRTUAL !Declare a virtual method
        END

ClassThree OneClass                        !Another Instance of a OneClass object
ClassFour  TwoClass                        !Another Instance of a TwoClass object

CODE
OneClass.BaseProc(1)    !BaseProc calls OneClass.Proc
TwoClass.BaseProc(2)   !BaseProc calls TwoClass.Proc
ClassThree.BaseProc(3) !BaseProc calls OneClass.Proc
ClassFour.BaseProc(4)  !BaseProc calls TwoClass.Proc

OneClass.BaseProc PROCEDURE(REAL Parm)
CODE
SELF.Proc(Parm)        !Calls virtual method, either OneClass.Proc
                       ! TwoClass.Proc, depending on which
                       ! class instance is executing

```

See Also:

CLASS

Procedure Overloading

Procedure Overloading means allowing multiple PROCEDURE definitions to use the same name. This is one form of polymorphism. In order to allow this each PROCEDURE using a shared name must receive different parameters so the compiler can decide, based on the parameters passed which PROCEDURE to call.

The idea here is to allow more than one procedure of the same name, but with different prototypes, so separate (but usually similar) operations can occur on different data types. From an efficiency viewpoint, Procedure Overloading is much more efficient than coding a single procedure with omissible parameters, for those cases where you may or may not receive multiple parameters.

The Clarion language also allows polymorphic procedures through the use of the ? and *? parameters, but Procedure Overloading extends this polymorphic ability to also include Entity-parameters and "named group" parameters.

One example of Procedure Overloading is the Clarion OPEN statement, which initializes an entity for use in the program. Depending on what type of entity is passed to it (a FILE, a WINDOW, a VIEW, etc.), it performs related but physically different functions.

See Also:

Rules for Procedure Overloading

Name Mangling and C++ Compatibility

Rules for Procedure Overloading

The Clarion language has built-in data type conversion which can make overload resolution difficult for the compiler. Therefore, there are rules governing how the compiler resolves functional overloading, which are applied in the following order:

1. Entity-parameters are resolved to FILE, KEY, WINDOW, and QUEUE. If a prototype can be chosen on the basis of these alone then the compiler does (most of the Clarion built in procedures fall into this category). Note that KEY and VIEW are implicitly derived from FILE, just as APPLICATION and REPORT are implicitly derived from WINDOW.
2. All "named group" parameters must match a group of their own structure. Procedure-parameters are matched by structure. CLASSES must match by name, not simply by structure.
3. A prototype must match in the number and placement of non-omittable parameters. This is the third criteria (not the first) so that the compiler can usually guess which prototype the user was aiming at and give a more meaningful error message.
4. If there are no matching prototypes then allow derivation. At this point a KEY would be allowed to match a FILE and a group that is derived would match one of its base classes. If one level of derivation does not work, the compiler continues up the tree. All QUEUES now match QUEUE and GROUP etc. CLASSES derive before other parameter types.
5. Variable-parameters (unnamed) must exactly match the actual data type passed. A *GROUP matches a *STRING. Any variable-parameter matches *?.
6. All Value-parameters are considered to have the same type.

Example:

```
MAP
Func  PROCEDURE(WINDOW)      ! 1
Func  PROCEDURE(FILE)       ! 2
Func  PROCEDURE(KEY)        ! 3
Func  PROCEDURE(FILE,KEY)   ! 4
Func  PROCEDURE(G1)         ! 5
Func  PROCEDURE(G0)         ! 6
Func  PROCEDURE(KEY,G0)     ! 7
Func  PROCEDURE(FILE,G1)    ! 8
Func  PROCEDURE(SHORT = 10) ! 9
Func  PROCEDURE(LONG)       ! 10
Func  PROCEDURE()           ! Illegal, indistinguishable from 9
Func  PROCEDURE(*SHORT)     ! 11
Func1  PROCEDURE(*SHORT)
Func1a PROCEDURE(*SHORT)
Func2  PROCEDURE(*LONG)
Func  PROCEDURE(Func1)      ! 12
Func  PROCEDURE(Func1a)    ! Illegal, same as 12
Func  PROCEDURE(Func2)     ! 13
END
```

```
G0 GROUP
  END
G1 GROUP(G0)
  END

CODE
Func(A:Window)    ! Calls 1 by rule 1
Func(A:File)      ! Calls 2 by rule 1
Func(A:Key)       ! Calls 3 by rule 1
Func(A:View)      ! Calls 2 by rule 4
Func(A:Key,A:Key)! Calls 4 by rule 4 (would call key,key if present)
Func(A:G0)        ! Calls 6 by rule 2
Func(A:G1)        ! Calls 5 by rule 2
Func(A:Func2)     ! Calls 13 by rule 2
Func(A:Key,A:G1) ! Error - Ambiguous. If rule 4 is used then 7 & 8 are both possible
Func(A:Short)     ! Error - Ambiguous. Calls 9 or 11
Func(A:Real)      ! Calls 9 by rule 6
Func              ! Calls 9 by rule 3
```

See Also:

CLASS

Name Mangling and C++ Compatibility

Each overloaded function will have a link-time name composed of the procedure label and a "mangled" argument list (the NAME attribute can be used to disable name mangling). This is designed so that some degree of cross-calling between C++ and Clarion is possible. On the C++ side you need:

```
#pragma name(prefix=>"")
```

and the name in all caps. On the Clarion side you need a MODULE structure with a null string as its parameter:

```
MODULE('')
END
```

The only procedures that can be cross-called are those whose prototypes only contain data types from the following list. Clarion Variable-parameters (passed by address) correspond to reference parameters on the C side unless they are omissible, in which case they correspond to pointer parameters.

<u>Clarion</u>	<u>C++</u>
BYTE	unsigned char
USHORT	unsigned short
SHORT	short
LONG	long
ULONG	unsigned long
SREAL	float
REAL	double
*CSTRING (with RAW)	char&
<*CSTRING> (with RAW)	char*
<*GROUP> (with RAW)	void*

Note that for C++ compatibility the return type of a PROCEDURE is not mangled into the name. A corollary effect is that procedures cannot be distinguished by return type.

Example:

```
//C++ prototypes:
#pragma name(prefix=>"")
void HADD(short,short);
void HADD(long*,unsigned char);
void HADD(short unsigned &);
void HADD(char *,void *);

!Clarion prototypes:
MODULE('')
  hADD(short,short)
  HaDD(<*long>,byte)
  HAdD(*ushort)
  HADd(<*CSTRING>,<*GROUP>),RAW
END
```

See Also: NAME

Compiler Directives

Compiler Directives are statements that tell the compiler to take some action at compile time. These statements are not included in the executable program object code which the compiler generates. Therefore, there is no run-time overhead associated with their use.

ASSERT (set assumption for debugging)

ASSERT(*expression*, [*message*])

ASSERT Specifies an assumption for debugging purposes.

expression A Boolean expression that *should* always evaluate as true (any value other than blank or zero).

message An optional string expression (up to 64K) which displays in the dialog window.

The **ASSERT** statement specifies an *expression* to evaluate at the exact point in the program where the ASSERT is placed. This may be any kind of Boolean expression and should be formulated such that the expected evaluation result is always true (any value other than blank or zero). The purpose of ASSERT is to catch erroneous assumptions for the programmer.

If debug is on and the *expression* is false (blank or zero), an error message displays indicating the specific line number and source code module where the asserted *expression* was false. The user is invited to GPF the program at that point, which allows Clarion's post-mortem debuggers to activate.

If debug is off, the *expression* is still evaluated, but no error message is displayed if the result is false. To activate error messages in release build (debug is off), you can add the following project define to your application:

```
asserts=>on
```

Example:

```
MyQueue QUEUE
F1      LONG
      END

CODE
LOOP X# = 1 TO 10
  MyQueue.F1 = X#
  ADD(MyQueue)
  ASSERT(~ERRORCODE(), 'ADD MyQueue Error ` & ERROR()')
END
LOOP X# = 1 TO 10
  GET(MyQueue, X#)
  ASSERT(~ERRORCODE())      !This error only happens if the ADD above fails
END

*****
!- Get Single Configuration Record
Access:CONFIG.Open()
SET(CONFIG)
ASSERT(~Access:Config.Next(), 'Config record missing')
```

BEGIN (define code structure)**BEGIN***statements***END**

BEGIN Declares a single code statement structure.*statements* Executable program instructions.

The **BEGIN** compiler directive tells the compiler to treat the *statements* as a single structure. The **BEGIN** structure must be terminated by a period or the **END** statement.

BEGIN is usually used in an **EXECUTE** control structure to allow several lines of code to be treated as one.

Example:

```
EXECUTE Value
  Proc1      !Execute if Value = 1
  BEGIN      !Execute if Value = 2
    Value += 1
    Proc2
  END
  Proc3      !Execute if Value = 3
END
```

See Also:

EXECUTE

COMPILE (specify source to compile)

COMPILE(*terminator* [,*expression*])

COMPILE	Specifies a block of source code lines to be included in the compilation.
<i>terminator</i>	A string constant that marks the last line of a block of source code.
<i>expression</i>	An expression allowing conditional execution of the COMPILE. The expression is either an EQUATE whose value is zero or one, or EQUATE = integer.

The **COMPILE** directive specifies a block of source code lines to be included in the compilation. The included block begins with the COMPILE directive and ends with the line that contains the same string constant as the *terminator*. The entire terminating line is included in the COMPILE block.

The optional *expression* parameter permits conditional COMPILE. The form of the *expression* is fixed. It is the label of an EQUATE statement, or a Conditional Switch set in the Project System, and may be followed by an equal sign (=) and an integer constant.

The code between COMPILE and the *terminator* is compiled only if the *expression* is true. If the *expression* contains an EQUATE that has not yet been defined, then the referenced EQUATE is assumed to be zero (0).

Although the *expression* is not required, COMPILE without an *expression* parameter is not necessary because all source code is compiled unless explicitly omitted. COMPILE and OMIT are opposites.

Example:

```

    OMIT('***',_WIDTH32_)           !OMIT only if application is 32-bit
SIGNED   EQUATE(SHORT)
UNSIGNED EQUATE(USHORT)
***
    COMPILE('***',_WIDTH32_)       !COMPILE only if application is 32-bit
SIGNED   EQUATE(LONG)
UNSIGNED EQUATE(ULONG)
***

COMPILE('EndOfFile',OnceOnly = 0) !COMPILE only the first time encountered because the
OnceOnly EQUATE(1)                ! OnceOnly EQUATE is defined after the COMPILE that
                                   ! references it, so a second pass during the same
                                   ! compilation will not re-compile the code

Demo EQUATE(1)                     !Specify the Demo EQUATE value
CODE
COMPILE('EndDemoChk',Demo = 1)    !COMPILE only if Demo equate is turned on
  DO DemoCheck                    !Check for demo limits passed
  ! EndDemoChk                    !End of conditional COMPILE code
! EndOfFile

!The following example below shows how OMIT and COMPILE can be nested
COMPILE ('**32bit**',_width32_) !outer COMPILE

  COMPILE ('*debug*',_debug_)
  DEBUGGER::BUTTONLIST Equate('&Continue|&Halt|&Debug')
  !end- COMPILE ('*debug*',_debug_)

  OMIT ('*debug*',_debug_)
  DEBUGGER::BUTTONLIST Equate('&Continue|&Halt')
  !end- OMIT ('*debug*',_debug_)

!end- COMPILE ('**32bit**',_width32_) !end outer COMPILE

OMIT ('**32bit**',_width32_)
  DEBUGGER::BUTTONLIST Equate('&Continue|&Halt')
!end- OMIT ('**32bit**',_width32_)

```

See Also:

OMIT

EQUATE

INCLUDE (compile code in another file)**INCLUDE**(*filename* [,*section*]) [,ONCE]

INCLUDE	Specifies source code to be compiled which exists in a separate file which is not a MEMBER module.
<i>filename</i>	A string constant that contains the DOS file specification for a source file. If the extension is omitted, .CLW is assumed.
<i>section</i>	A string constant which is the <i>string</i> parameter of the SECTION directive marking the beginning of the source code to be included.
ONCE	The ONCE attribute precludes any INCLUDED data from being compiled more than once, which can result in compile warnings or errors. Regarding the use of the <i>section</i> attribute, ONCE is applied on the entire <i>filename</i> , so subsequent uses of INCLUDE(<i>filename</i> , <i>section</i>) will be ignored.

The **INCLUDE** directive specifies source code to be compiled which exists in a separate file which is not a MEMBER module. Starting on the line of the INCLUDE directive, the source file, or the specified *section* of that file, is compiled as if it appeared in sequence within the source module being compiled. You can nest INCLUDEs up to 3 deep, so you can INCLUDE a file that includes a file that includes a file but that latter file must not include anything.

The compiler uses the Redirection file (*CurrentReleaseName*.RED) to find the file, searching the path specified for that type of *filename* (usually by extension). This makes it unnecessary to provide a complete path in the *filename* to be included. A discussion of the Redirection file is in the *User's Guide* and the *Project System* chapter of the *Programmer's Guide*.

Example:

```
GenLedger PROCEDURE      !Declare procedure
  INCLUDE('filedefs.clw') !Include file definitions here
  CODE                   !Begin code section
  INCLUDE('Setups','ChkErr') !Include error check from setups.clw
```

See Also:

SECTION

ONCE

EQUATE (assign label)

		<i>label</i>	
label	EQUATE ([<i>constant</i>])
		<i>picture</i>	
		<i>type</i>	

EQUATE	Assigns a label to another label or constant.
<i>label</i>	The <i>label</i> of any statement preceding the EQUATE statement. This is used to declare an alternate statement label. This may not be the label of a PROCEDURE or ROUTINE statement.
<i>constant</i>	A numeric or string <i>constant</i> . This is used to declare a shorthand label for a constant value. It also makes a constant easy to locate and change. This may be omitted only in an ITEMIZE structure. A constant expression may also be used (like 1+2, or BOR(1111b,0001b)).
<i>picture</i>	A <i>picture</i> token. This is used to declare a shorthand label for a picture token. However, the screen and report formatter in the Clarion Editor will not recognize the equated label as a valid picture.
<i>type</i>	A data type. This is usually used to declare a single method of declaring a variable as one of several data types, depending upon compiler settings (like a C++ typedef for a simple data type).

The **EQUATE** directive assigns a label to another label or constant. It does not use any run-time memory. The label of an EQUATE directive cannot be the same as its parameter.

Example:

```
Init      EQUATE(SetupProg)      !Set alias label
Off       EQUATE(0)              !Off means zero
On        EQUATE(1)              !On means one
PI        EQUATE(3.1415927)      !The value of PI
EnterMsg  EQUATE('Press Ctrl-Enter to SAVE')
SocSecPic EQUATE(@P###-##-####P) !Soc-sec number picture
```

See Also:

Reserved Words

ITEMIZE

ITEMIZE (enumeration data structure)

```
[label] ITEMIZE( [ seed ] ) [,PRE( )]
```

```
    equates
```

```
END
```

label An optional label for the ITEMIZE structure.

ITEMIZE An enumeration data structure.

seed An integer constant or constant expression specifying the value of the first EQUATE in the structure.

PRE Declare a label prefix for variables within the structure.

equates Multiple consecutive EQUATE declarations which specify positive integer values in the range 0 to 65,535.

An **ITEMIZE** structure declares an enumerated data structure. If the first *equate* does not declare a value and there is no *seed* value specified, its value is one (1). All following *equates* following the first increment by one (1) if no value is specified for the subsequent *equate*. If a value is specified on a subsequent *equate*, all *equates* following that continue incrementing by one (1) from the specified value.

Equates within the ITEMIZE structure are referenced by prepending the prefix to the label of the *equate* (PRE attribute--PRE:EquateLabel). If the ITEMIZE structure has an empty prefix, then the *equates* are referenced by prepending the ITEMIZE *label* to the label of the *equate* (*label*:EquateLabel). If there is no prefix or *label*, then the *equates* are referenced by their own label without a prefix.

Example:

```
      ITEMIZE
False EQUATE(0)      !False = 0
True  EQUATE        !True = 1
      END

Color ITEMIZE(0),PRE !Seed value is zero
Red   EQUATE        !Color:Red = 0
White EQUATE        !Color:White = 1
Blue  EQUATE        !Color:Blue = 2
Pink  EQUATE(5)     !Color:Pink = 5
Green EQUATE        !Color:Green = 6
Last  EQUATE
      END

Stuff ITEMIZE(Color>Last + 1),PRE(My) !Constant expression as seed
X     EQUATE        !My:X = Color>Last + 1
Y     EQUATE        !My:Y = Color>Last + 2
Z     EQUATE        !My:Z = Color>Last + 3
      END
```

See Also:

EQUATE, PRE

OMIT (specify source not to be compiled)

OMIT(*terminator* [,*expression*])

OMIT	Specifies a block of source code lines to be omitted from the compilation.
<i>terminator</i>	A string constant that marks the last line of a block of source code.
<i>expression</i>	An expression allowing conditional execution of the OMIT. The expression is either an EQUATE whose value is zero or one, or EQUATE = integer.

The **OMIT** directive specifies a block of source code lines to be omitted from the compilation. These lines may contain source code comments or a section of code that has been "stubbed out" for testing purposes. The omitted block begins with the OMIT directive and ends with the line that contains the same string constant as the *terminator*. The entire terminating line is included in the OMIT block.

The optional *expression* parameter permits conditional OMIT. The form of the *expression* is fixed. It is the label of an EQUATE statement, or a Conditional Switch set in the Project System, and may be followed by an equal sign (=) and an integer constant.

The OMIT directive executes only if the *expression* is true. Therefore, the code between OMIT and the *terminator* is compiled only if the *expression* is not true. If the *expression* contains an EQUATE that has not yet been defined, then the referenced EQUATE is assumed to be zero (0). COMPILE and OMIT are opposites.

Example:

```

OMIT('**END**')                !Unconditional OMIT
!Main Program Loop
**END**
OMIT('***',_WIDTH32_)         !OMIT only if application is 32-bit
SIGNED EQUATE(SHORT)
***
COMPILE('***',_WIDTH32_)      !COMPILE only if application is 32-bit
SIGNED EQUATE(LONG)
***
OMIT('EndOfFile',OnceOnly)    !Compile only the first time encountered because the
OnceOnly EQUATE(1)           ! OnceOnly EQUATE is defined after the COMPILE that
                             ! references it, so a second pass during the same
                             ! compilation will not re-compile the code

Demo EQUATE(0)                !Specify the Demo EQUATE value
CODE
OMIT('EndDemoChk',Demo = 0) !OMIT only if Demo is turned off
  DO DemoCheck                !Check for demo limits passed
  !EndDemoChk                 !End of omitted code
! EndOfFile
!The following example below shows how OMIT and COMPILE can be nested
COMPILE ('**32bit**',_width32_) !outer COMPILE

  COMPILE ('*debug*',_debug_)
  DEBUGGER::BUTTONLIST Equate('&Continue|&Halt|&Debug')
  !end- COMPILE ('*debug*',_debug_)

  OMIT ('*debug*',_debug_)
  DEBUGGER::BUTTONLIST Equate('&Continue|&Halt')
  !end- OMIT ('*debug*',_debug_)

!end- COMPILE ('**32bit**',_width32_) !end outer COMPILE

OMIT ('**32bit**',_width32_)
  DEBUGGER::BUTTONLIST Equate('&Continue|&Halt')
!end- OMIT ('**32bit**',_width32_)

```

See Also:

COMPILE

EQUATE

ONCE (prevent duplicate included data)

The **ONCE** attribute precludes any INCLUDED data from being compiled more than once resulting in compile warnings or errors.

Example:

```
INCLUDE( 'KEYCODES.CLW' ), ONCE
```

See Also:

INCLUDE

SECTION (specify source code section)

SECTION(*string*)

SECTION Identifies the beginning of a block of executable source code or data declarations.

string A string constant which names the SECTION.

The **SECTION** compiler directive identifies the beginning of a block of executable source code or data declarations which may be INCLUDED in source code in another file. The SECTION's *string* parameter is used as an optional parameter of the INCLUDE directive to include a specific block of source code. A SECTION is terminated by the next SECTION or the end of the file.

Example:

```
SECTION('FirstSection')      !Begin section

FieldOne STRING(20)
FieldTwo LONG

SECTION('SecondSection')    !End previous section, begin new section

IF Number <> SavNumber
  DO GetNumber
END

SECTION('ThirdSection')    !End previous section, begin new section

CASE Action
OF 1
  DO AddRec
OF 2
  DO ChgRec
OF 3
  DO DelRec
END                          !Third section ends at end of file
```

See Also:

INCLUDE

SIZE (memory size in bytes)

	<i>variable</i>	
SIZE (<i>constant</i>)
	<i>picture</i>	

SIZE Supplies the amount of memory used for storage.

variable The label of a previously declared variable.

constant A numeric or string constant.

picture A picture token.

SIZE directs the compiler to supply the amount of memory (in bytes) used to store the *variable*, *constant*, or *picture*.

Example:

```
SavRec  STRING(1),DIM(SIZE(Cus:Record)      !Dimension the string to size of record
StringVar STRING(SIZE('SoftVelocity'))    !A string long enough for the constant
LOOP I# = 1 TO SIZE(ParseString)          !Loop for number of bytes in the string
PicLen = SIZE(@P(###)###-####P)         !Save size of the picture
```

See Also: LEN

3 - Variable Declarations

Simple Data Types

BYTE (one-byte unsigned integer)

label **BYTE**(*initial value*) [,DIM()] [,OVER()] [,NAME()] [,EXTERNAL] [,DLL] [,STATIC] [,THREAD]
 [,AUTO] [,PRIVATE] [,PROTECTED]

BYTE A one-byte unsigned integer.

Format: magnitude
 | |
 Bits: 7 0
 Range: 0 to 255

initial value A numeric constant. If omitted, the initial value is zero, unless the AUTO attribute is present.

DIM Dimension the variable as an array.

OVER Share a memory location with another variable.

NAME Specify an alternate, "external" name for the field.

EXTERNAL Specify the variable is defined, and its memory is allocated, in an external library. Not valid within FILE, QUEUE, or GROUP declarations.

DLL Specify the variable is defined in a .DLL. This is required in addition to the EXTERNAL attribute.

STATIC Specify the variable's memory is permanently allocated.

THREAD Specify memory for the variable is allocated once for each execution thread. Also implicitly adds the STATIC attribute on Procedure Local data.

AUTO Specify the variable has no *initial value*.

PRIVATE Specify the variable is not visible outside the module containing the CLASS methods. Valid only in a CLASS.

PROTECTED Specify the variable is not visible outside base CLASS and derived CLASS methods. Valid only in a CLASS.

BYTE declares a one-byte unsigned integer.

Example:

```
Count1 BYTE                !Declare one byte integer
Count2 BYTE,OVER(Count1) !Declare OVER the one byte integer
Count4 BYTE,DIM(5)        !Declare as a 5 element array
Count4 BYTE(5)           !Declare with initial value
```

SHORT (two-byte signed integer)

label **SHORT**([*initial value*]) [,DIM()] [,OVER()] [,NAME()] [,EXTERNAL] [,DLL] [,STATIC]
 [,THREAD] [,AUTO] [,PRIVATE] [,PROTECTED]

SHORT	A two-byte signed integer. Format: ± magnitude Bits: 15 14 0 Range: -32,768 to 32,767
<i>initial value</i>	A numeric constant. If omitted, the initial value is zero, unless the AUTO attribute is present.
DIM	Dimension the variable as an array.
OVER	Share a memory location with another variable.
NAME	Specify an alternate, "external" name for the field.
EXTERNAL	Specify the variable is defined, and its memory is allocated, in an external library. Not valid within FILE, QUEUE, or GROUP declarations.
DLL	Specify the variable is defined in a .DLL. This is required in addition to the EXTERNAL attribute.
STATIC	Specify the variable's memory is permanently allocated.
THREAD	Specify memory for the variable is allocated once for each execution thread. Also implicitly adds the STATIC attribute on Procedure Local data.
AUTO	Specify the variable has no <i>initial value</i> .
PRIVATE	Specify the variable is not visible outside the module containing the CLASS methods. Valid only in a CLASS.
PROTECTED	Specify the variable is not visible outside base CLASS and derived CLASS methods. Valid only in a CLASS.

SHORT declares a two-byte signed integer, using the Intel 8086 word integer format. The high-order bit of this configuration is the sign bit (0 = positive, 1 = negative). Negative values are represented in standard two's complement notation.

Example:

```
Count1 SHORT                                !Declare two-byte signed integer
Count2 SHORT,OVER(Count1)                  !Declare OVER the two-byte signed integer
Count3 SHORT,DIM(4)                        !Declare it an array of 4 shorts
Count4 SHORT(5)                            !Declare with initial value
Count5 SHORT,EXTERNAL                      !Declare as external
Count6 SHORT,EXTERNAL,DLL                  !Declare as external in a .DLL
Count7 SHORT,NAME('SixCount')             !Declare with external name
ExampleFile FILE,DRIVER('Clarion')        !Declare a file
Record      RECORD
Count7      SHORT,NAME('Counter') !Declare with external name
           END
           END
```

USHORT (two-byte unsigned integer)

label **USHORT**([*initial value*]) [, **DIM**()] [, **OVER**()] [, **NAME**()] [, **EXTERNAL**] [, **DLL**] [, **STATIC**]
 [, **THREAD**] [, **AUTO**] [, **PRIVATE**] [, **PROTECTED**]

USHORT A two-byte unsigned integer.

Format: magnitude

||

Bits: 15 0

Range: 0 to 65,535

initial value A numeric constant. If omitted, the initial value is zero, unless the **AUTO** attribute is present.

DIM Dimension the variable as an array.

OVER Share a memory location with another variable.

NAME Specify an alternate, "external" name for the field.

EXTERNAL Specify the variable is defined, and its memory is allocated, in an external library. Not valid within **FILE**, **QUEUE**, or **GROUP** declarations.

DLL Specify the variable is defined in a .DLL. This is required in addition to the **EXTERNAL** attribute.

STATIC Specify the variable's memory is permanently allocated.

THREAD Specify memory for the variable is allocated once for each execution thread. Also implicitly adds the **STATIC** attribute on Procedure Local data.

AUTO Specify the variable has no *initial value*.

PRIVATE Specify the variable is not visible outside the module containing the **CLASS** methods. Valid only in a **CLASS**.

PROTECTED Specify the variable is not visible outside base **CLASS** and derived **CLASS** methods. Valid only in a **CLASS**.

USHORT declares a two-byte unsigned integer in the Intel 8086 word format. There is no sign bit in this configuration.

Example:

```
Count1 USHORT                                !Declare two-byte unsigned integer
Count2 USHORT,OVER(Count1)                  !Declare OVER the two-byte unsigned integer
Count3 USHORT,DIM(4)                        !Declare it an array of 4 unsigned shorts
Count4 USHORT(5)                            !Declare with initial value
Count5 USHORT,EXTERNAL                      !Declare as external
Count6 USHORT,EXTERNAL,DLL                 !Declare as external in a .DLL
Count7 USHORT,NAME('SixCount')            !Declare with external name
ExampleFile FILE,DRIVER('Btrieve')        !Declare a file
Record      RECORD
Count8      USHORT,NAME('Counter') !Declare with external name
           END
           END
```


Example:

```
Count1 LONG                                !Declare four-byte signed integer
Count2 LONG,OVER(Count1)                   !Declare OVER the four-byte signed integer
Count3 LONG,DIM(4)                          !Declare it an array of 4 longs
Count4 LONG(5)                              !Declare with initial value
Count5 LONG,EXTERNAL                        !Declare as external
Count6 LONG,EXTERNAL,DLL                    !Declare as external in a .DLL
Count7 LONG,NAME('SixCount')               !Declare with external name
ExampleFile FILE,DRIVER('Clarion')         !Declare a file
Record      RECORD
Count8      LONG,NAME('Counter')           !Declare with external name
           END
           END
```


Example:

```
Count1 ULONG                                !Declare four-byte unsigned integer
Count2 ULONG,OVER(Count1)                  !Declare OVER four-byte unsigned integer
Count3 ULONG,DIM(4)                        !Declare it an array of 4 unsigned longs
Count4 ULONG(5)                            !Declare with initial value
Count5 ULONG,EXTERNAL                      !Declare as external
Count6 ULONG,EXTERNAL,DLL                 !Declare as external in a .DLL
Count7 ULONG,NAME('SixCount')            !Declare with external name
ExampleFile FILE,DRIVER('Btrieve')       !Declare a file
Record      RECORD
Count8      ULONG,NAME('Counter') !Declare with external name
           END
           END
```

SIGNED (signed integer)

label **SIGNED**([*initial value*]) [,**DIM**()] [,**OVER**()] [,**NAME**()] [,**EXTERNAL**] [,**DLL**] [,**STATIC**] [,**THREAD**]
 [,**AUTO**] [,**PRIVATE**] [,**PROTECTED**]

SIGNED	A signed integer defined as a LONG.
<i>initial value</i>	A numeric constant. If omitted, the initial value is zero, unless the AUTO attribute is present.
DIM	Dimension the variable as an array.
OVER	Share a memory location with another variable.
NAME	Specify an alternate, "external" name for the field.
EXTERNAL	Specify the variable is defined, and its memory is allocated, in an external library. Not valid within FILE, QUEUE, or GROUP declarations.
DLL	Specify the variable is defined in a .DLL. This is required in addition to the EXTERNAL attribute.
STATIC	Specify the variable's memory is permanently allocated.
THREAD	Specify memory for the variable is allocated once for each execution thread. Also implicitly adds the STATIC attribute on Procedure Local data.
AUTO	Specify the variable has no <i>initial value</i> .
PRIVATE	Specify the variable is not visible outside the module containing the CLASS methods. Valid only in a CLASS.
PROTECTED	Specify the variable is not visible outside base CLASS and derived CLASS methods. Valid only in a CLASS.

SIGNED declares a signed integer which is a LONG.

The SIGNED data type is most useful for prototyping Windows API calls.

Example:

```
Count1 SIGNED !Declares a LONG
```


SREAL (four-byte signed floating point)

label **SREAL**(*[initial value]*) [**DIM**()] [**OVER**()] [**NAME**()] [**EXTERNAL**] [**DLL**] [**STATIC**]
 [**THREAD**] [**AUTO**] [**PRIVATE**] [**PROTECTED**]

SREAL A four-byte floating point number.
 Format: ± exponent significand
 | | | |
 Bits: 31 30 23 0
 Range: 0, ± 1.175494e-38 .. ± 3.402823e+38 (6 significant digits)

initial value A numeric constant. If omitted, the initial value is zero, unless the AUTO attribute is present.

DIM Dimension the variable as an array.

OVER Share a memory location with another variable.

NAME Specify an alternate, "external" name for the field.

EXTERNAL Specify the variable is defined, and its memory is allocated, in an external library. Not valid within FILE, QUEUE, or GROUP declarations.

DLL Specify the variable is defined in a .DLL. This is required in addition to the EXTERNAL attribute.

STATIC Specify the variable's memory is permanently allocated.

THREAD Specify memory for the variable is allocated once for each execution thread. Also implicitly adds the STATIC attribute on Procedure Local data.

AUTO Specify the variable has no *initial value*.

PRIVATE Specify the variable is not visible outside the module containing the CLASS methods. Valid only in a CLASS.

PROTECTED Specify the variable is not visible outside base CLASS and derived CLASS methods. Valid only in a CLASS.

SREAL declares a four-byte floating point signed numeric variable, using the Intel 8087 short real (single precision) format.

Example:

```
Count1 SREAL                                !Declare four-byte signed floating point
Count2 SREAL,OVER(Count1)                   !Declare OVER the four-byte
                                             !signed floating point
Count3 SREAL,DIM(4)                          !Declare it an array of 4 floats
Count4 SREAL(5)                              !Declare with initial value
Count5 SREAL,EXTERNAL                        !Declare as external
Count6 SREAL,EXTERNAL,DLL                    !Declare as external in a .DLL
Count7 SREAL,NAME('SixCount')               !Declare with external name

ExampleFile FILE,DRIVER('Btrieve')          !Declare a file
Record      RECORD
Count8      SREAL,NAME('Counter')           !Declare with external name
           END
           END
```

REAL (eight-byte signed floating point)

label **REAL**([*initial value*]) [,**DIM**()] [,**OVER**()] [,**NAME**()] [,**EXTERNAL**] [,**DLL**] [,**STATIC**] [,**THREAD**]
 [,**AUTO**] [,**PRIVATE**] [,**PROTECTED**]

REAL	An eight-byte floating point number. <pre> Format: ± exponent significand Bits:63 62 52 0 Range: 0, ± 2.225073858507201e-308 .. ± 1.79769313496231e+308 (15 significant digits) </pre>
<i>initial value</i>	A numeric constant. If omitted, the initial value is zero, unless the AUTO attribute is present.
DIM	Dimension the variable as an array.
OVER	Share a memory location with another variable.
NAME	Specify an alternate, "external" name for the field.
EXTERNAL	Specify the variable is defined, and its memory is allocated, in an external library. Not valid within FILE, QUEUE, or GROUP declarations.
DLL	Specify the variable is defined in a .DLL. This is required in addition to the EXTERNAL attribute.
STATIC	Specify the variable's memory is permanently allocated.
THREAD	Specify memory for the variable is allocated once for each execution thread. Also implicitly adds the STATIC attribute on Procedure Local data.
AUTO	Specify the variable has no <i>initial value</i> .
PRIVATE	Specify the variable is not visible outside the module containing the CLASS methods. Valid only in a CLASS.
PROTECTED	Specify the variable is not visible outside base CLASS and derived CLASS methods. Valid only in a CLASS.

REAL declares an eight-byte floating point signed numeric variable, using the Intel 8087 long real (double precision) format.

Example:

```
Count1 REAL                                !Declare eight-byte signed floating point
Count2 REAL,OVER(Count1)                  !Declare OVER the eight-byte
                                           !signed floating point
Count3 REAL,DIM(4)                        !Declare it an array of 4 reals
Count4 REAL(5)                            !Declare with initial value
Count5 REAL,EXTERNAL                      !Declare as external
Count6 REAL,EXTERNAL,DLL                  !Declare as external in a .DLL
Count7 REAL,NAME('SixCount')             !Declare with external name
ExampleFile FILE,DRIVER('Clarion')       !Declare a file
Record      RECORD
Count8      REAL,NAME('Counter')         !Declare with external name
                                           END
                                           END
```

BFLOAT4 (four-byte signed floating point)

label **BFLOAT4**(*initial value*) [,**DIM**()] [,**OVER**()] [,**NAME**()] [,**EXTERNAL**] [,**DLL**] [,**STATIC**]
 [,**THREAD**] [,**AUTO**] [,**PRIVATE**] [,**PROTECTED**]

BFLOAT4 A four-byte floating point number.

Format:	exponent	±	significand

Bits:	31	23	22 0
Range:	0, ± 5.87747e-39 .. ± 1.70141e+38 (6 significant digits)		

initial value A numeric constant. If omitted, the initial value is zero, unless the **AUTO** attribute is present.

DIM Dimension the variable as an array.

OVER Share a memory location with another variable.

NAME Specify an alternate, "external" name for the field.

EXTERNAL Specify the variable is defined, and its memory is allocated, in an external library. Not valid within **FILE**, **QUEUE**, or **GROUP** declarations.

DLL Specify the variable is defined in a **.DLL**. This is required in addition to the **EXTERNAL** attribute.

STATIC Specify the variable's memory is permanently allocated.

THREAD Specify memory for the variable is allocated once for each execution thread. Also implicitly adds the **STATIC** attribute on Procedure Local data.

AUTO Specify the variable has no *initial value*.

PRIVATE Specify the variable is not visible outside the module containing the **CLASS** methods. Valid only in a **CLASS**.

PROTECTED Specify the variable is not visible outside base **CLASS** and derived **CLASS** methods. Valid only in a **CLASS**.

BFLOAT4 declares a four-byte floating point signed numeric variable, using the Microsoft BASIC (single precision) format. This data type is normally used for compatibility with existing data since it is internally converted to a **REAL** before all arithmetic operations.

Example:

```
Count1 BFLOAT4                                !Declare four-byte signed floating point
Count2 BFLOAT4,OVER(Count1)                   !Declare OVER the four-byte
                                                ! signed floating point
Count3 BFLOAT4,DIM(4)                          !Declare array of 4 single-precision reals
Count4 BFLOAT4(5)                              !Declare with initial value
Count5 BFLOAT4,EXTERNAL                        !Declare as external
Count6 BFLOAT4,EXTERNAL,DLL                    !Declare as external in a .DLL
Count7 BFLOAT4,NAME('SixCount')               !Declare with external name
ExampleFile FILE,DRIVER('Btrieve')           !Declare a file
Record          RECORD
Count8          BFLOAT4,NAME('Counter')        !Declare with external name
              END
              END
```


Example:

```
Count1 BFLOAT8                               !Declare eight-byte signed floating point
Count2 BFLOAT8,OVER(Count1)                  !Declare OVER
Count3 BFLOAT8,DIM(4)                         !Declare it an array of 4 reals
Count4 BFLOAT8(5)                             !Declare with initial value
Count5 BFLOAT8,EXTERNAL                       !Declare as external
Count6 BFLOAT8,EXTERNAL,DLL                   !Declare as external in a .DLL
Count7 BFLOAT8,NAME('SixCount')              !Declare with external name
ExampleFile FILE,DRIVER('Btrieve')           !Declare a file
Record          RECORD
Count8          BFLOAT8,NAME('Counter')       !Declare with external name
              END
              END
```

DECIMAL (signed packed decimal)

label **DECIMAL**(*length* [,*places*] [,*initial value*]) [,**DIM**()] [,**OVER**()] [,**NAME**()] [,**EXTERNAL**] [,**DLL**]
 [,**STATIC**] [,**THREAD**] [,**AUTO**] [,**PRIVATE**] [,**PROTECTED**]

DECIMAL	A packed decimal floating point number. <pre> Format: ± magnitude Bits: 127 124 0 Range: -9,999,999,999,999,999,999,999,999 to +9,999,999,999,999,999,999,999,999 </pre>
<i>length</i>	A required numeric constant containing the total number of decimal digits (integer and fractional portion combined) in the variable. The maximum <i>length</i> is 31.
<i>places</i>	A numeric constant that fixes the number of decimal digits in the fractional portion (to the right of the decimal point) of the variable. It must be less than or equal to the <i>length</i> parameter. If omitted, the variable will be declared as an integer.
<i>initial value</i>	A numeric constant. If omitted, the initial value is zero, unless the AUTO attribute is present.
DIM	Dimension the variable as an array.
OVER	Share a memory location with another variable.
NAME	Specify an alternate, "external" name for the field.
EXTERNAL	Specify the variable is defined, and its memory is allocated, in an external library. Not valid within FILE, QUEUE, or GROUP declarations.
DLL	Specify the variable is defined in a .DLL. This is required in addition to the EXTERNAL attribute.
STATIC	Specify the variable's memory is permanently allocated.
THREAD	Specify memory for the variable is allocated once for each execution thread. Also implicitly adds the STATIC attribute on Procedure Local data.
AUTO	Specify the variable has no <i>initial value</i> .
PRIVATE	Specify the variable is not visible outside the module containing the CLASS methods. Valid only in a CLASS.
PROTECTED	Specify the variable is not visible outside base CLASS and derived CLASS methods. Valid only in a CLASS.

DECIMAL declares a variable length packed decimal signed numeric variable. Each byte of a DECIMAL holds two decimal digits (4 bits per digit). The left-most byte holds the sign in its high-order nibble (0 = positive, anything else is negative) and one decimal digit. Therefore, DECIMAL variables always contain a fixed "odd" number of digits (DECIMAL(10) and DECIMAL(11) both use 6 bytes).

Example:

```
Count1 DECIMAL(5,0)                !Declare three-byte signed packed decimal
Count2 DECIMAL(5),OVER(Count1)     !Declare OVER the three-byte
                                   !signed packed decimal

Count3 DECIMAL(5,0),DIM(4)         !Declare it an array of 4 decimals
Count4 DECIMAL(5,0,5)              !Declare with initial value
Count5 DECIMAL(5,0),EXTERNAL       !Declare as external
Count6 DECIMAL(5,0),EXTERNAL,DLL   !Declare as external in a .DLL
Count7 DECIMAL(5,0),NAME('SixCount') !Declare with external name
ExampleFile FILE,DRIVER('TopSpeed') !Declare a file
Record      RECORD
Count8      DECIMAL(5,0),NAME('Counter') !Declare with external name
           END
           END
```


PDECIMAL declares a variable length packed decimal signed numeric variable in the Btrieve and IBM/EBCDIC type of format. Each byte of an PDECIMAL holds two decimal digits (4 bits per digit). The right-most byte holds the sign in its low-order nibble (0Fh or 0Ch = positive, 0Dh = negative) and one decimal digit. Therefore, PDECIMAL variables always contain a fixed "odd" number of digits (PDECIMAL(10) and PDECIMAL(11) both use 6 bytes).

Example:

```
Count1 PDECIMAL(5,0)                !Declare three-byte signed packed decimal
Count2 PDECIMAL(5),OVER(Count1)      !Declare OVER the three-byte
                                     ! signed packed decimal
Count3 PDECIMAL(5,0),DIM(4)          !Declare it an array of 4 decimals
Count4 PDECIMAL(5,0,5)               !Declare with initial value
Count5 PDECIMAL(5,0),EXTERNAL        !Declare as external
Count6 PDECIMAL(5,0),EXTERNAL,DLL    !Declare as external in a .DLL
Count7 PDECIMAL(5,0),NAME('SixCount') !Declare with external name
ExampleFile FILE,DRIVER('Btrieve')  !Declare a file
Record      RECORD
Count8      PDECIMAL(5,0),NAME('Counter') !Declare with external name
           END
           END
```

STRING (fixed-length string)

```

          | length      |
label STRING(      | string constant  |) [, DIM( )][, OVER( )] [, NAME( )] [, EXTERNAL] [, DLL]
[STATIC]
          | picture      |      [, THREAD] [, AUTO] [, PRIVATE] [, PROTECTED]

```

STRING	A character string. Format: A fixed number of bytes. Size: 4MB
<i>length</i>	A numeric constant that defines the number of bytes in the STRING. String variables are not initialized unless given a <i>string constant</i> .
<i>string constant</i>	The initial value of the STRING. The length of the STRING (in bytes) is set to the length of the <i>string constant</i> .
<i>picture</i>	Used to format the values assigned to the STRING. The length is the number of bytes needed to contain the formatted STRING.
DIM	Dimension the variable as an array.
OVER	Share a memory location with another variable.
NAME	Specify an alternate, "external" name for the field.
EXTERNAL	Specify the variable is defined, and its memory is allocated, in an external library. Not valid within FILE, QUEUE, or GROUP declarations.
DLL	Specify the variable is defined in a .DLL. This is required in addition to the EXTERNAL attribute.
STATIC	Specify the variable's memory is permanently allocated.
THREAD	Specify memory for the variable is allocated once for each execution thread. Also implicitly adds the STATIC attribute on Procedure Local data.
AUTO	Specify the variable has no <i>initial value</i> .
PRIVATE	Specify the variable is not visible outside the module containing the CLASS methods. Valid only in a CLASS.
PROTECTED	Specify the variable is not visible outside base CLASS and derived CLASS methods. Valid only in a CLASS.

STRING declares a fixed-length character string. The memory assigned to the STRING is initialized to all blanks unless the AUTO attribute is present.

In addition to its explicit declaration, all STRING variables are also implicitly declared as `STRING(1),DIM(length of string)`. This allows each character in the STRING to be addressed as an array element. If the STRING also has a DIM attribute, this implicit array declaration is the last (optional) dimension of the array (to the right of the explicit dimensions).

You may also directly address multiple characters within a STRING using the "string slicing" technique. This technique performs similar action to the SUB function, but is much more flexible and efficient (but does no bounds checking). It is more flexible because a "string slice" may be used on both the destination and source sides of an assignment statement and the SUB function can only be used as the source. It is more efficient because it takes less memory than individual character assignments or the SUB function.

To take a "slice" of the STRING, the beginning and ending character numbers are separated by a colon (:) and placed in the implicit array dimension position within the square brackets ([]) of the STRING. The position numbers may be integer constants, variables, or expressions. If variables are used, there must be at least one blank space between the variable name and the colon separating the beginning and ending number (to prevent PREFIX confusion).

Example:

```

Name          STRING(20)                !Declare 20 byte name field
ArrayString   STRING(5),DIM(20)         !Declare array
Company       STRING('SoftVelocity Corporation') !The software company - 20 bytes
Phone        STRING(@P(###)###-###P)    !Phone number field - 13 bytes
ExampleFile   FILE,DRIVER('Clarion')     !Declare a file
Record        RECORD
NameField     STRING(20),NAME('Name')    !Declare with external name
            END
            END

CODE
NameField = 'Tammi'                    !Assign a value
NameField[5] = 'y'                     ! change fifth letter
NameField[5:6] = 'ie'                  ! and change a "slice"
            ! the fifth and sixth letters

ArrayString[1] = 'First'                !Assign value to first element
ArrayString[1,2] = 'u'                  !Change first element 2nd character
ArrayString[1,2:3] = NameField[5:6]    !Assign slice to slice

```

ASTRING (atomic string)

label **ASTRING**([*stringtoken*] [,**DIM**()] [,**NAME**()] [,**EXTERNAL**] [,**DLL**] [,**STATIC**] [,**THREAD**]
 [,**AUTO**] [,**PRIVATE**] [,**PROTECTED**])

ASTRING	A reference to a character string.
<i>stringtoken</i>	The initial string token of the ASTRING.
DIM	Dimension the variable as an array.
OVER	The ASTRING reference may share a memory location with another variable.
NAME	Specify an alternate, "external" name for the field.
EXTERNAL	Specify the variable is defined, and its memory is allocated, in an external library.
DLL	Specify the variable is defined in a .DLL. This is required in addition to the EXTERNAL attribute.
STATIC	Specifies the memory for the ASTRING reference variable is permanently allocated.
THREAD	Specify memory for the variable is allocated once for each execution thread. Also implicitly adds the STATIC attribute on Procedure Local data.
AUTO	Specify the variable has no initial string token.
PRIVATE	Specify the variable is not visible outside the module containing the CLASS methods. Valid only in a CLASS.
PROTECTED	Specify the variable is not visible outside base CLASS and derived CLASS methods. Valid only in a CLASS.

ASTRING (similar to a Win32 ATOM) declares a reference to a variable length string (string token) with a maximum size of 64k. Although the size of the string token may be up to a maximum of 64k, it is recommended for use on smaller strings. The storage space for an ASTRING is allocated dynamically as needed for the lifetime of the procedure in which it is created. However the allocated storage is never deallocated; it is instead reused for subsequent use of the same text values.

ASTRINGs are useful when a lot of the same text is being stored and compared. Each time a new ASTRING is created, an entry is created in a memory table. When an ASTRING is assigned a value that already exists in the table, the ASTRING simply points to the existing ASTRING. This saves memory and makes string comparison very fast.

ASTRINGs are not supported by any of the file drivers.

ASTRINGs may be passed as parameters to procedures by value or by reference. They may also be passed as an untyped value or untyped variable parameters. An ASTRING can be returned from a procedure only as a value.

Example prototypes:

```
PROC1(ASTRING a)
PROC2(*ASTRING a)
PROC3(? a)
PROC4(*? a)
PROC5(),ASTRING
```

Example:

```
PROGRAM
```

```
MAP
END
```

```
FLAG LONG
```

```
AS1 ASTRING,OVER(FLAG)
```

```
AS2 ASTRING
```

```
CODE
```

```
AS1 = 'SoftVelocity' ! storage is allocated for the string 'SoftVelocity'
```

```
AS2 = 'SoftVelocity' ! storage is not allocated again, instead AS1 and
! AS2 share the same reference value.
```

```
AS2 = 'Hello' ! new storage allocated for the new text string
! Hello
```

```
AS2 = 'SoftVelocity' ! No new storage is allocated, the reference for
! AS1 now equals AS2 again
```

```
IF FLAG = 0
```

```
MESSAGE('AS1 is NULL')
```

```
ELSE
```

```
MESSAGE('AS1 = ' & AS1)
```

```
END
```

CSTRING (fixed-length null terminated string)

```

        | length      |
label  CSTRING(    |string constant  ) [, DIM( )] [, OVER( )] [, NAME( )] [, EXTERNAL] [, DLL]
        | picture    |          [, STATIC] [, THREAD] [, AUTO] [, PRIVATE] [, PROTECTED]

```

CSTRING	A character string. Format: A fixed number of bytes. Size: Unlimited
<i>length</i>	A numeric constant that defines the number of bytes of storage the string will use. This must include a position for the terminating null character. String variables are not initialized unless given a <i>string constant</i> .
<i>string constant</i>	A string constant containing the initial value of the string. The length of the string is set to the length of the <i>string constant</i> plus the terminating null character.
<i>picture</i>	The picture token used to format the values assigned to the string. The length of the string is the number of bytes needed to contain the formatted string and the terminating null character.
DIM	Dimension the variable as an array.
OVER	Share a memory location with another variable.
NAME	Specify an alternate, "external" name for the field.
EXTERNAL	Specify the variable is defined, and its memory is allocated, in an external library. Not valid within FILE, QUEUE, or GROUP declarations.
DLL	Specify the variable is defined in a .DLL. This is required in addition to the EXTERNAL attribute.
STATIC	Specify the variable's memory is permanently allocated.
THREAD	Specify memory for the variable is allocated once for each execution thread. Also implicitly adds the STATIC attribute on Procedure Local data.
AUTO	Specify the variable has no <i>initial value</i> .
PRIVATE	Specify the variable is not visible outside the module containing the CLASS methods. Valid only in a CLASS.
PROTECTED	Specify the variable is not visible outside base CLASS and derived CLASS methods. Valid only in a CLASS.

CSTRING declares a character string terminated by a null character (ASCII zero). The memory assigned to the CSTRING is initialized to a zero length string unless the AUTO attribute is present.

CSTRING matches the string data type used in the "C" language and the "ZSTRING" data type of the Btrieve Record Manager. Storage and memory requirements are fixed-length, however the terminating null character is placed at the end of the data entered. CSTRING should be used to achieve compatibility with outside files or procedures.

In addition to its explicit declaration, all CSTRINGs are implicitly declared as a `STRING(1),DIM(length of string)`. This allows each character in the CSTRING to be addressed as an array element. If the CSTRING also has a DIM attribute, this implicit array declaration is the last (optional) dimension of the array (to the right of the explicit dimensions).

You may also directly address multiple characters within a CSTRING using the "string slicing" technique. This technique performs similar action to the SUB function, but is much more flexible and efficient (but does no bounds checking). It is more flexible because a "string slice" may be used on both the destination and source sides of an assignment statement and the SUB function can only be used as the source. It is more efficient because it takes less memory than individual character assignments or the SUB function.

To take a "slice" of the CSTRING, the beginning and ending character numbers are separated by a colon (:) and placed in the implicit array dimension position within the square brackets ([]) of the CSTRING. The position numbers may be integer constants, variables, or expressions. If variables are used, there must be at least one blank space between the variable name and the colon separating the beginning and ending number (to prevent PREFIX confusion).

Since a CSTRING must be null-terminated, the programmer must be responsible for ensuring that an ASCII zero is placed at the end of the data if the field is only accessed through its array elements or as a "slice" (not as a whole entity). Also, a CSTRING can have "junk" stored after the null terminator. Because of this they do not work well inside GROUPs.

Example:

```

Name          CSTRING(21)                !Declare 21 byte field - 20 bytes data
OtherName     CSTRING(21),OVER(Name)      !Declare field over name field
Contact       CSTRING(21),DIM(4)         !Array 21 byte fields - 80 bytes data
Company       CSTRING('SoftVelocity Corporation') !21 byte string - 20 bytes data
Phone        CSTRING(@P(###)###-####P)  !Declare 14 bytes - 13 bytes data
ExampleFile   FILE,DRIVER('Btrieve')    !Declare a file
Record        RECORD
NameField     CSTRING(21),NAME('ZstringField') !Declare with external name
              END
              END

CODE
Name = 'Tammi'                !Assign a value
Name[5] = 'y'                 ! then change fifth letter
Name[6] = 's'                 ! then add a letter
Name[7] = '<0>'                ! and handle null terminator
Name[5:6] = 'ie'              ! and change a "slice"
                              ! -- the fifth and sixth letters

Contact[1] = 'First'          !Assign value to first element
Contact[1,2] = 'u'            !Change first element 2nd character
Contact[1,2:3] = Name[5:6]    !Assign slice to slice

```

PSTRING (embedded length-byte string)

```

          | length      |
label PSTRING(      | string constant  | ) [, DIM( ) ] [, OVER( ) ] [, NAME( ) ] [, EXTERNAL] [, DLL]
[STATIC]
          | picture     |      [, THREAD] [, AUTO] [, PRIVATE] [, PROTECTED]

```

PSTRING	A character string. Format: A fixed number of bytes. Size: 2 to 256 bytes.
<i>length</i>	A numeric constant that defines the number of bytes in the string. This must include the length-byte.
<i>string constant</i>	A string constant containing the initial value of the string. The length of the string is set to the length of the <i>string constant</i> plus the length-byte.
<i>picture</i>	The picture token used to format the values assigned to the string. The length of the string is the number of bytes needed to contain the formatted string plus the first position length byte. String variables are not initialized unless given a <i>string constant</i> .
DIM	Dimension the variable as an array.
OVER	Share a memory location with another variable.
NAME	Specify an alternate, "external" name for the field.
EXTERNAL	Specify the variable is defined, and its memory is allocated, in an external library. Not valid within FILE, QUEUE, or GROUP declarations.
DLL	Specify the variable is defined in a .DLL. This is required in addition to the EXTERNAL attribute.
STATIC	Specify the variable's memory is permanently allocated.
THREAD	Specify memory for the variable is allocated once for each execution thread. Also implicitly adds the STATIC attribute on Procedure Local data.
AUTO	Specify the variable has no <i>initial value</i> .
PRIVATE	Specify the variable is not visible outside the module containing the CLASS methods. Valid only in a CLASS.
PROTECTED	Specify the variable is not visible outside base CLASS and derived CLASS methods. Valid only in a CLASS.

PSTRING declares a character string with a leading length byte included in the number of bytes declared for the string. The memory assigned to the PSTRING is initialized to a zero length string unless the AUTO attribute is present. PSTRING matches the string data type used by the Pascal language and the "LSTRING" data type of the Btrieve Record Manager. Storage and memory requirements are fixed-length, however, the leading length byte will contain the number of characters actually stored. PSTRING is internally converted to a STRING intermediate value for string operations during program execution. PSTRING should be used to achieve compatibility with outside files or procedures.

In addition to its explicit declaration, all PSTRINGs are implicitly declared as a PSTRING(1),DIM(*length of string*). This allows each character in the PSTRING to be addressed as an array element. If the PSTRING also has a DIM attribute, this implicit array declaration is the last (optional) dimension of the array (to the right of the explicit dimensions).

You may also directly address multiple characters within a PSTRING using the "string slicing" technique. This technique performs similar action to the SUB function, but is much more flexible and efficient (but does no bounds checking). It is more flexible because a "string slice" may be used on both the destination and source sides of an assignment statement and the SUB function can only be used as the source. It is more efficient because it takes less memory than individual character assignments or the SUB function. To take a "slice" of the PSTRING, the beginning and ending character numbers are separated by a colon (:) and placed in the implicit array dimension position within the square brackets ([]) of the PSTRING. The position numbers may be integer constants, variables, or expressions. If variables are used, there must be at least one blank space between the variable name and the colon separating the beginning and ending number (to prevent PREFIX confusion).

Since a PSTRING must have a leading length byte, the programmer must be responsible for ensuring that its value is always correct if the field is only accessed through its array elements or as a "slice" (not as a whole entity). The PSTRING's length byte is addressed as element zero (0) of the array (BLOB and PSTRING are the only exceptions in Clarion where an array has a zero element). Therefore, the valid range of array indexes for a PSTRING(30) would be 0 to 29. Also, a PSTRING can have 'junk' stored outside the active portion of the string. Because of this they do not work well inside GROUPs.

Example:

```
Name      PSTRING(21)                !Declare 21 byte field - 20 bytes data
OtherName PSTRING(21),OVER(Name)    !Declare field over name field
Contact   PSTRING(21),DIM(4)       !Array 21 byte fields - 80 bytes data
Company   PSTRING('SoftVelocity Corporation') !21 byte string - 20 bytes data
Phone     PSTRING(@P(###)###-####P) !Declare 14 bytes - 13 bytes data
ExampleFile FILE,DRIVER('Btrieve') !Declare a file
Record    RECORD
NameField PSTRING(21),NAME('LstringField') !Declare with external name
          END
          END

CODE
Name = 'Tammi'           !Assign a value
Name[5] = 'y'           ! then change fifth letter
Name[6] = 's'           ! then add a letter
Name[0] = '<6>'          ! and handle length byte
Name[5:6] = 'ie'        ! and change a "slice" -- the 5th and 6th letters
Contact[1] = 'First'    !Assign value to first element
Contact[1,2] = 'u'      !Change first element 2nd character
Contact[1,2:3] = Name[5:6] !Assign slice to slice
```

Implicit String Arrays and String Slicing

In addition to their explicit declaration, all STRING, CSTRING and PSTRING variables have an implicit array declaration of one character strings, dimensioned by the length of the string. This is directly equivalent to declaring a second variable as:

```
StringVar  STRING(10)
StringArray STRING(1),DIM(SIZE(StringVar)),OVER(StringVar)
```

This implicit array declaration allows each character in the string to be directly addressed as an array element, without the need of the second declaration. The PSTRING's length byte is addressed as element zero (0) of the array, as is the first byte of a BLOB (the only two cases in Clarion where an array has a zero element).

If the string also has a DIM attribute, this implicit array declaration is the last (optional) dimension of the array (to the right of the explicit dimensions). The MAXIMUM procedure does not operate on the implicit dimension, you should use SIZE instead.

You may also directly address multiple characters within a string using the "string slicing" technique. This technique performs a similar function to the SUB procedure, but is much more flexible and efficient (but does no bounds checking). It is more flexible because a "string slice" may be used as either the *destination* or *source* sides of an assignment statement, while the SUB procedure can only be used as the source. It is more efficient because it takes less memory than either individual character assignments or the SUB procedure.

To take a "slice" of the string, the beginning and ending character numbers are separated by a colon (:) and placed in the implicit array dimension position within the square brackets ([]) of the string. The position numbers may be integer constants, variables, or expressions (internally computed as LONG base type). If variables are used, there must be at least one blank space between the variable name and the colon separating the beginning and ending number (to prevent PREFIX confusion).

Example:

```
Name      STRING(15)
CONTACT   STRING(15),DIM(4)
CODE
Name = 'Tammi'           !Assign a value
Name[5] = 'y'           ! then change fifth letter
Name[6] = 's'           ! then add a letter
Name[0] = '<6>'          ! and handle length byte
Name[5:6] = 'ie'        ! and change a "slice" -- the fifth and sixth letters
Contact[1] = 'First'    !Assign value to first element
Contact[1,2] = 'u'      !Change first element 2nd character
Contact[1,2:3] = Name[5:6] !Assign slice to first element 2nd & 3rd characters
```

See Also: STRING, CSTRING, PSTRING, BLOB

Example:

```
DueDate      DATE                !Declare a date field
OtherDate    DATE,OVER(DueDate)  !Declare field over date field
ContactDate  DATE,DIM(4)         !Array of 4 date fields
ExampleFile  FILE,DRIVER('Btrieve') !Declare a file
Record       RECORD
DateRecd     DATE,NAME('DateField') !Declare with external name
            END
            END
```

See Also:

Standard Date

TIME (four-byte time)

label **TIME** [,DIM()] [,OVER()] [,NAME()] [,EXTERNAL] [,DLL] [,STATIC] [,THREAD] [,AUTO]
 [,PRIVATE] [,PROTECTED]

TIME	A four-byte time. Format: hh mm ss hs Bits: 31 23 15 7 0 Range: hours: 0 to 23 minutes: 0 to 59 seconds: 0 to 59 seconds/100: 0 to 99
DIM	Dimension the variable as an array.
OVER	Share a memory location with another variable.
NAME	Specify an alternate, "external" name for the field.
EXTERNAL	Specify the variable is defined, and its memory is allocated, in an external library. Not valid within FILE, QUEUE, or GROUP declarations.
DLL	Specify the variable is defined in a .DLL. This is required in addition to the EXTERNAL attribute.
STATIC	Specify the variable's memory is permanently allocated.
THREAD	Specify memory for the variable is allocated once for each execution thread. Also implicitly adds the STATIC attribute on Procedure Local data.
AUTO	Specify the variable has no <i>initial value</i> .
PRIVATE	Specify the variable is not visible outside the module containing the CLASS methods. Valid only in a CLASS.
PROTECTED	Specify the variable is not visible outside base CLASS and derived CLASS methods. Valid only in a CLASS.

TIME declares a four byte time variable. This format matches the "TIME" field type used by the Btrieve Record Manager. A TIME used in a numeric expression is converted to the number of hundredths of a second elapsed since midnight (Clarion Standard Time - usually stored as a LONG). TIME fields should be used to achieve compatibility with outside files or procedures.

Example:

```
CheckoutTime  TIME                !Declare checkout time field
OtherTime     TIME,OVER(CheckoutTime) !Declare field over time field
ContactTime   TIME,DIM(4)          !Array of 4 time fields
ExampleFile   FILE,DRIVER('Btrieve') !Declare a file
Record        RECORD
TimeRecd      TIME,NAME('TimeField') !Declare with external name
              END
              END
```

See Also:

Standard Time

Use of CLEAR() and reference assignments with QUEUE entries.

Once an ANY variable in a QUEUE has been assigned a value, another simple assignment statement will assign a new value to the ANY. This means the previous value is replaced by the new value. If the first value has already been added to the QUEUE, then that entry will "point at" a value that no longer exists.

Once an ANY variable in a QUEUE has been reference assigned a variable (AnyVar &= SomeVariable), another reference assignment statement will assign a new variable to the ANY. This means the previous "pointer" is disposed of and replaced by the new "pointer." If the first reference has already been added to the QUEUE, then that entry will "point at" a "pointer" that no longer exists.

In both cases, the QUEUE record (or fields of ANY type only) must be CLEAR()ed before setting new values for the next ADD() or PUT().

In addition, you need to reference assign a NULL to the ANY variable (AnyVar &= NULL), prior to deleting the QUEUE entry, in order to avoid memory leaks.

As explained above, the ANY variable maintains its own data area where it keeps the value or "pointer" to the referenced variable. An assignment of a new value to the ANY variable using the &= operator disposes its previous internal data. The CLEAR(*variable*) statement sets the memory location occupied by the ANY variable to NULL, *without* disposing of its internal data.

Example:

```

MyQueue QUEUE
AnyField ANY                !Declare a variable to contain any value
Type STRING(1)
END
DueDate DATE                !Declare a date field
CODE
MyQueue.AnyField = 'SoftVelocity' !Assign a string value
MyQueue.Type = 'S'           !Flag data type
ADD(MyQueue)
CLEAR(MyQueue)              !Clear the reference
MyQueue.AnyField &= DueDate  !Assign a Reference to a DATE
MyQueue.Type = 'R'          !Flag data type
ADD(MyQueue)
MyQueue.AnyField &= NULL     !Reference assign NULL to clear
LOOP X# = RECORDS(MyQueue) TO 1 BY -1 !Process the QUEUE
  GET(MyQueue,X#)
  ASSERT(~ERRORCODE())
  CASE MyQueue.Type
  OF 'S'
    DO StringRoutine
  OF 'R'
    DO ReferenceRoutine
  END
  MyQueue.AnyField &= NULL     !Reference assign NULL before deleting
  DELETE(MyQueue)
  ASSERT(~ERRORCODE())
END

```

See Also:

Simple Assignment Statements

Reference Assignment Statements

LIKE (inherited data type)

new declaration **LIKE**(*like declaration*) [,**DIM**()] [,**OVER**()] [,**PRE**()] [,**NAME**()] [,**EXTERNAL**] [,**DLL**]
 [,**STATIC**] [,**THREAD**] [,**BINDABLE**]

LIKE	Declares a variable whose data type is inherited from another variable.
<i>new declaration</i>	The label of the new data element declaration.
<i>like declaration</i>	The label of the data element declaration whose definition will be used. This may be any simple data type, or a reference to any simple data type (except &STRING), or the label of a GROUP or QUEUE structure.
DIM	Dimension the variables into an array.
OVER	Share a memory location with another variable or structure.
PRE	Declare a label prefix for variables within the <i>new declaration</i> structure (if the <i>like declaration</i> is a complex data structure). This is not required, since you may use the <i>new declaration</i> in the Field Qualification syntax to directly reference any member of the new structure.
NAME	Specify an alternate, "external" name for the field.
EXTERNAL	Specify the variable is defined, and its memory is allocated, in an external library. Not valid within FILE, QUEUE, or GROUP declarations.
DLL	Specify the variable is defined in a .DLL. This is required in addition to the EXTERNAL attribute.
STATIC	Specify the variable's memory is permanently allocated.
THREAD	Specify memory for the variable is allocated once for each execution thread. Also implicitly adds the STATIC attribute on Procedure Local data.
BINDABLE	Specify all variables in the group may be used in dynamic expressions.

LIKE tells the compiler to define the *new declaration* using the same definition as the *like declaration*, including all attributes. If the original *like declaration* changes, so does the *new declaration*.

The *new declaration* may use the DIM and OVER attributes. If the *like declaration* has a DIM attribute, the *new declaration* is already an array. If a further DIM attribute is added to the *new declaration*, the array is further dimensioned.

The PRE and NAME attributes may be used, if appropriate. If the *like declaration* already has these attributes, the *new declaration* will inherit them and compiler errors can occur. To correct this, specify a PRE or NAME attribute on the *new declaration* to override the inherited attribute.

If the *like declaration* names a QUEUE, LIKE does not create a new QUEUE, because the *like declaration* is simply treated as a GROUP. The *like declaration* QUEUE is converted to a *new declaration* GROUP. The same is true if the *like declaration* is a RECORD structure. Similarly, if the *like declaration* is a MEMO, the *new declaration* becomes a STRING of the maximum size of the MEMO.

You may use LIKE to create a new instance of a CLASS. However, simply declaring the new instance by naming the CLASS as the data type performs an implicit LIKE. For either type of instance declaration, the DIM, OVER, PRE, and NAME attributes are invalid; all other attributes are valid for a CLASS instance declaration.

Example:

```

Amount      REAL                !Define a field
QTDAmount   LIKE(Amount)        !Use same definition
YTDAmount    LIKE(QTDAmount)     !Use same definition again
MonthlyAmts LIKE(Amount),DIM(12) !Use same definition for array, 12 elements
AmtPrPerson LIKE(MonthlyAmts),DIM(10)
                !Use same definition for array of 120 elements (12,10)

Construct GROUP                !Define a group
Field1      LIKE(Amount)       ! Construct.field1 - real
Field2      STRING(10)         ! Construct.field2 - string(10)
                END

NewGroup    LIKE(Construct)     !Define new group, containing
                ! NewGroup.field1 - real
                ! NewGroup.field2 - string(10)

MyQue      QUEUE                !Define a queue
Field1     STRING(10)
Field2     STRING(10)
                END

MyGroup    LIKE(MyQue)          !Define new GROUP, like the QUEUE

AmountFile FILE,DRIVER('Clarion'),PRE(Amt)
Record     RECORD
Amount     REAL                !Define a field
QTDAmount  LIKE(Amount)        !Use same definition
                END
                END

```

```
Animal    CLASS
Feed      PROCEDURE(short amount),VIRTUAL
Die       PROCEDURE
Age       LONG
Weight    LONG
          END
Cat       LIKE(Animal) !New instance of an Animal CLASS
Bird      Animal       !New instance of an Animal CLASS (implicit LIKE)
```

See Also:

DIM

OVER

PRE

NAME

Field Qualification

Implicit Variables

Implicit variables are not declared in data declarations. They are created by the compiler when it first encounters them. Implicit variables are automatically initialized to blank or zero; they do not have to be explicitly assigned values before use. You may always assume that they contain blanks or zero before your program's first assignment to them. Implicit variables are generally used for: array subscripts, true/false switches, intermediate variables in complex calculations, loop counters, etc.

The Clarion language provides three types of implicit variables:

- # A label terminated by a # names an implicit LONG.
- \$ A label terminated by a \$ names an implicit REAL.
- " A label terminated by a " names an implicit STRING(32).

Any implicit variable used in the global data declaration area (between the keywords PROGRAM and CODE) is Global data, assigned static memory and visible throughout the program. Any implicit variable used between the keywords MEMBER and PROCEDURE is Module data, assigned static memory and visible only to the procedures defined in the module. Any other implicit variable is Local data, assigned dynamic memory on the program's stack and visible only in the procedure. Implicits used in ROUTINES incur more overhead than those not in ROUTINES, so should be used sparingly, if at all.

Since the compiler dynamically creates implicit variables as they are encountered, there is a danger that problems may arise that can be difficult to trace. This is due to the lack of compile-time error and type checking on implicit variables. For example, if you incorrectly spell the name of a previously used implicit variable, the compiler will not tell you, but will simply create a new implicit variable with the new spelling. When your program checks the value in the original implicit variable, it will be incorrect. Therefore, implicit variables should be used with care and caution, and only within a limited scope (or not at all).

Example:

```
LOOP Counter# = 1 TO 10           !Implicit LONG loop counter
  ArrayField[Counter#] = Counter# * 2 !to initialize an array
END
```

```
Address" = CLIP(City) & ', ' & State & ' ' & Zip
!Implicit STRING(32)
```

```
MESSAGE(Address")
!Used to display a temporary value
```

```
Percent$ = ROUND((Quota / Sales),.1) * 100
!Implicit REAL
```

```
MESSAGE(FORMAT(Percent$,@P%<<<.&##P))
!Used to display a temporary value
```

See Also:

Data Declarations and Memory Allocation

Reference Variables

A reference variable contains a reference to another data declaration (its "target"). You declare a reference variable by prepending an ampersand (&) to the data type of its target (such as, &BYTE, &FILE, &LONG, etc.) or by declaring an ANY variable. Depending upon the target's data type, the reference variable may contain the target's memory address, or a more complex internal data structure (describing the location and type of target data).

Valid reference variable declarations:

&BYTE	&SHORT	&USHORT	&LONG	&ULONG	&DATE	&TIME
&REAL	&SREAL	&BFLOAT8	&BFLOAT4	&DECIMAL	&PDECIMAL	&STRING
&CSTRING	&PSTRING	&GROUP	&QUEUE	&FILE	&KEY	&BLOB
&VIEW	&WINDOW	ANY				

The &STRING, &CSTRING, &PSTRING, &DECIMAL, and &PDECIMAL declarations do not require length parameters, since all the necessary information about the specific target data item is contained in the reference itself. This means a &STRING reference variable may contain a reference to any length STRING variable.

A reference variable declared as &WINDOW can target an APPLICATION, WINDOW, or REPORT structure. References to these structures are internally treated as the same by the Clarion runtime library.

An ANY variable can contain a reference to any of the simple data types, and so, is equivalent to any of the above except &GROUP, &QUEUE, &FILE, &KEY, &BLOB, &VIEW, and &WINDOW.

Reference Assignment

The &= operator executes a reference assignment statement (destination &= source) to assign the *source's* reference to the *destination* reference variable. You may also use a reference assignment statement in conditional expressions.

The NULL built-in variable is used to "un-reference" a reference variable or to detect an "un-referenced" reference variable in a conditional expression.

Reference Variable Usage

The label of a reference variable is syntactically correct every place in executable code where its target is allowed. This means that, any statement that takes the label of a WINDOW as a parameter can also take the label of an &WINDOW reference variable which has been reference-assigned a WINDOW structure.

When used in a code statement, the reference variable is automatically "dereferenced" to supply the statement with the value of its target. The only exception is reference assignment statements, when the reference assigns the reference to the data item it is referencing. For example:

```

Var1    LONG           !Var1 is a LONG
RefVar1 &LONG         !RefVar1 is a reference to a LONG
RefVar2 &LONG         !RefVar2 is also a reference to a LONG
CODE
RefVar1 &= Var1      !RefVar1 now references Var1
RefVar2 &= RefVar1  !RefVar2 now also references Var1
RefVar1 &= NULL     !RefVar1 now references nothing

```

Reference Variable Declarations

Reference variables may not be declared within FILE or VIEW structures, but they may be declared within GROUP, QUEUE, and CLASS structures. Issuing CLEAR(StructureName) for a GROUP, QUEUE, or CLASS structure containing a reference variable is equivalent to reference assigning NULL to the reference variable.

Global references cross thread boundaries, and so, may be used to reference data items in other execution threads.

Named QUEUE and CLASS References

In addition to the data types listed above, you may also have references to "named" QUEUES (&QueueName) and to named CLASSES (&ClassName). This allows you to use references to pass "named group" parameters, which allow the receiving procedure access to the component fields of the named structure.

A reference to a named QUEUE or CLASS may be a "forward reference." That is, the named QUEUE or CLASS does not have to have been declared previous to the reference variable declaration which "points at" it. However, the forward reference must be resolved before the reference variable can be used. In the case where the reference variable is contained within a CLASS declaration, the forward reference must be resolved before the object is instantiated, else the reference will be blank and unusable.

There are several advantages to using forward references. You can have a QUEUE of object references which each contains a reference to a QUEUE of object references which each contains a reference to a QUEUE of object references ... For example, you could create a queue of siblings within a CLASS structure like this:

```

FamilyQ      QUEUE
Sibling      &FamilyClass      !A forward reference
            END

FamilyClass CLASS
Family      &FamilyQ      !
            END

```

Another advantage is the ability to truly "hide" the targets of PRIVATE references in CLASS declarations. For example:

```

!An include file (MyFile.inc) contains:
WidgetManager CLASS,TYPE
WidgetList    &WidgetQ,PRIVATE !
DoSomething   PROCEDURE
            END

!Another file (MyFile.CLW) contains:
MEMBER('MyApp')
INCLUDE('MyFile.INC')

WidgetQ      QUEUE,TYPE
Widget      STRING(40)
WidgetNumber LONG
            END

MyWidget     WidgetManager !Actual instantiation must follow
            ! forward reference resolution
MyWidget.DoSomething PROCEDURE
CODE
SELF.WidgetList &= NEW(WidgetQ) !Valid code
SELF.WidgetList.Widget = 'Widget One'
SELF.WidgetList.WidgetNumber = 1
ADD(SELF.WidgetList)

```

In this example, references to SELF.WidgetList are valid only within the MyFile.CLW file.

Example:

```

App1  APPLICATION('Hello')
      END

App2  APPLICATION('Buenos Dias')
      END

AppRef &WINDOW !Reference to an APPLICATION, WINDOW, or REPORT

Animal  CLASS
Feed    PROCEDURE(SHORT amount),VIRTUAL
Die     PROCEDURE
Age     LONG
Weight  LONG
      END

Carnivore CLASS(Animal),TYPE
Feed    PROCEDURE(Animal)
      END

Cat     CLASS(Carnivore)
Feed    PROCEDURE(SHORT amount),VIRTUAL
Potty   BYTE
      END

Bird    Animal          !Instance of an Animal CLASS
AnimalRef &Animal      !Reference to an Animal CLASS

CODE
IF CTL:Language = 'Spanish' !If spanish language user
  AppRef &= App2           ! reference spanish application frame
ELSE
  AppRef &= App1           ! else reference english application frame
END
OPEN(AppRef)              !Open the referenced application frame window

IF SomeCondition
  AnimalRef &= Cat        !Reference the Cat
ELSE
  AnimalRef &= Bird      !Reference the Bird
END
AnimalRef.Feed(10)       !Feed whatever is referenced

```

See Also: Reference Assignment Statements, CLASS, GROUP, QUEUE, ANY

Data Declarations and Memory Allocation

Global, Local, Static, and Dynamic

Data declarations automatically allocate memory to store the data values. Global, Local, Static, and Dynamic are terms that describe types of memory allocation.

The terms "Global" and "Local" refer to the visibility of data (also known as its "scope"):

- "Global" means the data is visible to all procedures in the program.
- "Local" means the data has limited visibility. This may be limited to one PROCEDURE or ROUTINE, or limited to a specific set of procedures in a single source module.

The terms "Static" and "Dynamic" refer to the persistence of the data's memory allocation:

- "Static" means the data is allocated memory that is not released until the entire program is finished executing.
- "Dynamic" means the data is allocated memory at run time. Data declared locally for the PROCEDURE or ROUTINE is allocated on the stack of the thread called to that PROCEDURE or ROUTINE. The stack memory allocated for such data is released on returning to the caller.
- "Dynamic" also means that data is allocated by the program in the heap with the use of an explicit NEW, or implicitly, by some runtime library statements (assignment to ANY variables, ADDing to a QUEUE, etc.). Memory blocks allocated in the heap exist until their explicit releasing: DISPOSE for NEWed variables, FREE or DELETE for QUEUEs, etc.

Data Declaration Sections

There are three areas where data can be declared in a Clarion program:

- In the PROGRAM module, after the keyword PROGRAM and before the CODE statement. This is the **Global data** section.
- In a MEMBER module, after the keyword MEMBER and before the first PROCEDURE statement. This is the **Module data** section.
- In a PROCEDURE, after the keyword PROCEDURE and before the CODE statement. This is the **Local data** section.
- In a ROUTINE, after the keyword DATA and before the CODE statement. This is the **Routine Local data** section.

Global data is visible to executable statements and expressions in every PROCEDURE in the PROGRAM. Global data is always in scope. Global data is allocated Static memory and is available to every PROCEDURE in the PROGRAM.

Module data is visible only to the set of PROCEDURES contained in the MEMBER module. It may be passed as a parameter to PROCEDURES in other MEMBER modules, if required. Module data first comes into scope when any PROCEDURE in the MODULE is called. Module data is also allocated Static memory.

Local data is visible only within the PROCEDURE in which it is declared, or any Local Derived Methods declared within the PROCEDURE. Local data comes into scope when the PROCEDURE is called and goes out of scope when a RETURN statement (explicit or implicit) executes. It may be passed as a parameter to any other PROCEDURE. Local data is allocated Dynamic memory. The memory is allocated on the program's stack for variables smaller than the stack threshold (5K default), otherwise they are automatically placed onto the heap. This can be overridden by using the STATIC attribute, making its value persistent between calls to the procedure. FILE declarations are always allocated static memory (on the heap), even when declared in a Local Data section.

Dynamic memory allocation for Local data allows a PROCEDURE to be truly recursive, receiving a new copy of its local variables each time it is called.

Routine Local data is visible only within the ROUTINE in which it is declared. It may be passed as a parameter to any PROCEDURE. Routine Local data comes into scope when the ROUTINE is called and goes out of scope when an EXIT statement (explicit or implicit) executes. Routine Local data is allocated Dynamic memory. The memory is allocated on the program's stack for variables smaller than the stack threshold (5K default), otherwise they are automatically placed onto the heap. A ROUTINE has its own name scope, so the labels used for Routine Local data may duplicate variable names used in other ROUTINES or even the procedure containing the ROUTINE. Variables declared in a ROUTINE may not have the STATIC or THREAD attributes. See Also: PROGRAM, MEMBER, PROCEDURE, CLASS, PROCEDURE Prototypes, STATIC, THREAD

Variable Size Declarations

It is now possible to declare variables of STRING-like and DECIMAL/PDECIMAL types of variable size to the compiler.

Restrictions are the same as for variable-size arrays (see below): declarations are available in the procedure or routine local scope only, and all variables used in the expression must be known at the time of the variable's creation.

Example:

```
VarLength    LONG
VarString    STRING(VarLength)

CODE
  VarLength = 200
  VarString = `String of up to 200 characters`
```

Variable-size arrays

Consider the following example:

```
VariableArray ROUTINE

Element      LONG
DynArray     CSTRING(100),DIM(Element)      !declare a variable length array

CODE
  Element = 100                             !assign number of elements
```

There are 3 restrictions when using this technique:

- 1) The dimensioned variable (*Element*) must be locally declared in the respective PROCEDURE or ROUTINE data section and have no STATIC, THREAD or EXTERNAL attributes applied.
- 2) The dimensioned variable can not be a field component of any compound structure (GROUP, QUEUE, CLASS, RECORD)
- 3) All variables used in dimension expressions must be initialized at the moment of array initialization, i.e., they must be declared with an initial value before the array declaration, or they must be declared in outer scope and receive a value before the call to the PROCEDURE or ROUTINE, or they can be a parameter of the PROCEDURE.

NEW (allocate heap memory)

reference &= **NEW**(*datatype*)

<i>reference</i>	The label of a reference variable that matches the <i>datatype</i> .
NEW	Creates a new instance of the <i>datatype</i> on the heap.
<i>datatype</i>	The label of a previously declared CLASS or QUEUE structure, or any simple data type declaration. This may contain a variable as the parameter of the data type to allow truly dynamic declarations.

The **NEW** statement creates a new instance of the *datatype* on the heap. NEW is only valid on the *source* side of a reference assignment statement. Memory allocated by NEW is automatically initialized to blank or zero when allocated, and must be explicitly de-allocated with the DISPOSE statement (else you'll create a "memory leak").

Example:

```
StringRef &STRING           !A reference to any STRING variable
LongRef   &LONG             !A reference to any LONG variable

Animal CLASS
Feed     PROCEDURE(short amount)
Weight   LONG
        END
AnimalRef &Animal           !A reference to any Animal CLASS

NameQ QUEUE
Name     STRING(30)
        END
QueRef   &NameQ             !A reference to any QUEUE with only a STRING(30)

CODE
AnimalRef &= NEW(Animal)    !Create new instance of an Animal class

QueRef &= NEW(NameQ)        !Create new instance of a NameQ QUEUE

StringRef &= NEW(STRING(50)) !Create new STRING(50) variable

X# = 35                      !Assign 35 to a variable and then
StringRef &= NEW(STRING(X#)) ! use that variable to Create a new STRING(35)

LongRef &= NEW(LONG)        !Create new LONG variable
```

See Also:

DISPOSE

DISPOSE (de-allocate heap memory)**DISPOSE**(*reference*)

DISPOSE De-allocates heap memory previously allocated by a NEW statement.

reference The label of a reference variable previously used in a reference assignment with the NEW statement. This *reference* may be NULL and no ill effects will occur.

The **DISPOSE** statement de-allocates the heap memory previously allocated by a NEW statement. If DISPOSE is not called, the memory is not returned to the operating system for reuse (creating a "memory leak"). However, if you DISPOSE of a *reference* that is still in use (such as a QUEUE being displayed in a LIST control) you will quite likely cause a GPF that will be very difficult to track down.

DISPOSE(SELF) is a legal statement to de-allocate the current object instance. However, if used, it must be the last statement in the procedure, or any following references to the object will cause problems.

There is a way to pass a &STRING reference to a procedure in a way that it can be disposed in that procedure. Although a *STRING cannot be disposed, consider the following code:

```
MyProc PROCEDURE (*STRING S)
  Ref &STRING,AUTO
  CODE
  Ref &= S
  DISPOSE (Ref)
```

The Parameter/result of *STRING type can be considered as a read only &STRING value: you can change the string it points to but you can't set it to reference to another string. DISPOSE sets the reference passed to it as a parameter to NULL and hence *STRING parameters and results can't be DISPOSEd directly but assigning their reference to a reference variable gives a solution.

Example:

```
StringRef &STRING           !A reference to any STRING variable

Animal CLASS,TYPE
Feed  PROCEDURE(short amount),VIRTUAL
Weight LONG
      END
AnimalRef &Animal          !A reference to any Animal CLASS
```

```
NameQ  QUEUE
Name    STRING(30)
      END
QueRef  &NameQ          !A reference to any QUEUE with only a STRING(30)

CODE
AnimalRef &= NEW(Animal) !Create new instance of an Animal class
DISPOSE(AnimalRef)      !De-allocate the Animal

QueRef &= NEW(NameQ)     !Create new instance of a NameQ QUEUE
DISPOSE(QueRef)         !De-allocate the queue

StringRef &= NEW(STRING(50)) !Create new STRING(50) variable
DISPOSE(StringRef)      !De-allocate the STRING(50)
```

See Also:

NEW

Picture Tokens

Picture tokens provide a masking format for displaying and editing variables. There are seven types of picture tokens: numeric and currency, scientific notation, string, date, time, pattern, and key-in template.

Numeric and Currency Pictures

@N [*currency*] [*sign*] [*fill*] *size* [*grouping*] [*places*] [*sign*] [*currency*] [**B**]

@N	All numeric and currency pictures begin with @N.
<i>currency</i>	Either a dollar sign (\$) or any string constant enclosed in tildes (~). When it precedes the <i>sign</i> indicator and there is no <i>fill</i> indicator, the <i>currency</i> symbol "floats" to the left of the high order digit. If there is a <i>fill</i> indicator, the <i>currency</i> symbol remains fixed in the left-most position. If the <i>currency</i> indicator follows the <i>size</i> and <i>grouping</i> , it appears at the end of the number displayed.
<i>sign</i>	Specifies the display format for negative numbers. If a hyphen (-) precedes the <i>fill</i> and <i>size</i> indicators, negative numbers will display with a leading minus sign. If a hyphen follows the <i>size</i> , <i>places</i> , and <i>currency</i> indicators, negative numbers will display with a trailing minus sign. If parentheses are placed in both positions, negative numbers will be displayed enclosed in parentheses. To prevent ambiguity, a trailing minus <i>sign</i> should always have <i>grouping</i> specified.
<i>fill</i>	Specifies leading zeros, spaces, or asterisks (*) in any leading zero positions, and suppresses default <i>grouping</i> . If the <i>fill</i> is omitted, leading zeros are suppressed. <ul style="list-style-type: none"> 0 (zero) Produces leading zeroes _ (underscore) Produces leading spaces * (asterisk) Produces leading asterisks
<i>size</i>	The <i>size</i> is required to specify the total number of significant digits to display, including the number of digits in the <i>places</i> indicator and any formatting characters.

grouping A *grouping* symbol, other than a comma (the default), can appear right of the *size* indicator to specify a three digit group separator. To prevent ambiguity, a hyphen *grouping* indicator should also specify the *sign*.

- . (period) Produces periods
- (hyphen) Produces hyphens
- _ (underscore) Produces spaces

places Specifies the decimal separator symbol and the number of decimal digits. The number of decimal digits must be less than the *size*. The decimal separator may be a period (.), grave accent (`) (produces periods *grouping* unless overridden), or the letter "v" (used only for STRING field storage declarations--not for display).

- . (period) Produces a period
- ` (grave accent) Produces a comma
- v Produces no decimal separator

B Specifies blank display whenever its value is zero.

The numeric and currency pictures format numeric values for screen display or in reports. If the value is greater than the maximum value the picture can display, a string of pound signs (#) is displayed.

Example:

<u>Numeric</u>	<u>Result</u>	<u>Format</u>
@N9	4,550,000	Nine digits, group with commas (default)
@N_9B	4550000	Nine digits, no grouping, leading blanks if zero
@N09	004550000	Nine digits, leading zero
@N*9	***45,000	Nine digits, asterisk fill, group with commas
@N9_	4 550 000	Nine digits, group with spaces
@N9.	4.550.000	Nine digits, group with periods

<u>Decimal</u>	<u>Result</u>	<u>Format</u>
@N9.2	4,550.75	Two decimal places, period decimal separator
@N_9.2B	4550.75	Two decimal places, period decimal separator, no grouping, blank if zero
@N_9'2	4550,75	Two decimal places, comma decimal separator
@N9.'2	4.550,75	Comma decimal separator, group with periods
@N9_'2	4 550,75	Comma decimal separator, group with spaces

<u>Signed</u>	<u>Result</u>	<u>Format</u>
@N-9.2B	-2,347.25	Leading minus sign, blank if zero
@N9.2-	2,347.25-	Trailing minus sign
@N(10.2)	(2,347.25)	Enclosed in parens when negative

Dollar

<u>Currency</u>	<u>Result</u>	<u>Format</u>
@N\$9.2B	\$2,347.25	Leading dollar sign, blank if zero
@N\$10.2-	\$2,347.25-	Leading dollar sign, trailing minus when negative
@N\$(11.2)	\$(2,347.25)	Leading dollar sign, in parens when negative

International

<u>Currency</u>	<u>Result</u>	<u>Format</u>
@N12_'2~ F~	1 5430,50 F	France
@N~L. ~12'	L. 1.430.050	Italy
@N~£~12.2	£1,240.50	United Kingdom
@N~kr~12'2	kr1.430,50	Norway
@N~DM~12'2	DM1.430,50	Germany
@N12_'2~ mk~	1 430,50 mk	Finland
@N12'2~ kr~	1.430,50 kr	Sweden

Storage-Only Pictures:

```
Variable1 STRING(@N_6v2)           !Declare as 6 bytes stored without decimal
CODE
Variable1 = 1234.56                 !Assign value, stores '123456' in file
MESSAGE(FORMAT(Variable1,@N_7.2)) !Display with decimal point: '1234.56'
```

Scientific Notation Pictures

@Emsn[B]

@E	All scientific notation pictures begin with @E.
m	Determines the total number of characters in the format provided by the picture.
s	Specifies the decimal separation character, and the grouping character when the n value is greater than 3. . (period) period and comma .. (period period) period and period ' (grave accent) comma and period _.(underscore period) period and space
n	Indicates the number of digits that appear to the left of the decimal point.
B	Specifies that the format displays as blank when the value is zero.

The scientific notation picture formats very large or very small numbers. The format is a decimal number raised by a power of ten.

Example:

<u>Picture</u>	<u>Value</u>	<u>Result</u>
@E9.0	1,967,865	.20e+007
@E12.1	1,967,865	1.9679e+006
@E12.1B	0	
@E12.1	-1,967,865	-1.9679e+006
@E12.1	.000000032	3.2000e-008
@E12_.4	1,967,865	1 967.865e+003

String Pictures

@Slength

@S All string pictures begin with @S.

length Determines the number of characters in the picture format.

A string picture describes an unformatted string of a specific *length*.

Example:

Name `STRING(@S20)` !A 20 character string field

Date Pictures

@Dn [s] [*direction* [*range*]] [B]

@D	All date pictures begin with @D.
n	Determines the date picture format. Date picture formats range from 1 through 18. A leading zero (0) indicates a zero-filled day or month.
s	A separation character between the month, day, and year components. If omitted, the slash (/) appears. <ul style="list-style-type: none"> . (period) Produces periods ' (grave accent) Produces commas - (hyphen) Produces hyphens _ (underscore) Produces spaces
<i>direction</i>	A right or left angle bracket (> or <) that specifies the "Intellidate" direction (> indicates future, < indicates past) for the <i>range</i> parameter. Valid only on ENTRY date pictures with two-digit years.
<i>range</i>	An integer constant in the range of zero (0) to ninety-nine (99) that specifies the "Intellidate" century for the <i>direction</i> parameter. Valid only on ENTRY date pictures with two-digit years. If omitted, the default value is 80.
B	Specifies that the format displays as blank when the value is zero.

Dates may be stored in numeric variables (usually LONG), a DATE field (for Btrieve compatibility), or in a STRING declared with a date picture. A date stored in a numeric variable is called a "Clarion Standard Date." The stored value is the number of days since December 28, 1800. The date picture token converts the value into one of the date formats.

The century for dates in any picture with a two-digit year is resolved using "Intellidate" logic. Date pictures that do not specify *direction* and *range* parameters assume the date falls in the range of the next 19 or previous 80 years. The *direction* and *range* parameters allow you to change this default. The *direction* parameter specifies whether the *range* specifies the future or past value. The opposite *direction* then receives the opposite value (100-*range*) so that any two-digit year results in the correct century.

For example, the picture @D1>60 specifies using the appropriate century for each year 60 years in the future and 39 years in the past. If the current year is 1996, when the user enters "5/01/40," the date is in the year 2040, and when the user enters "5/01/60," the date is in the year 1960.

For those date pictures which contain month names, the actual names are customizable in an Environment file (.ENV). See the Internationalization section for more information.

Example:

<u>Picture</u>	<u>Format</u>	<u>Result</u>
@D1	mm/dd/yy	10/31/59
@D1>40	mm/dd/yy	10/31/59
@D01	mm/dd/yy	01/01/95
@D2	mm/dd/yyyy	10/31/1959
@D3	mmm dd,yyyy	OCT 31,1959
@D4	mmmmmmmmmm dd, yyyy	October 31, 1959
@D5	dd/mm/yy	31/10/59
@D6	dd/mm/yyyy	31/10/1959
@D7	dd mmm yy	31 OCT 59
@D8	dd mmm yyyy	31 OCT 1959
@D9	yy/mm/dd	59/10/31
@D10	yyyy/mm/dd	1959/10/31
@D11	yyymmdd	591031
@D12	yyyymmdd	19591031
@D13	mm/yy	10/59
@D14	mm/yyyy	10/1959
@D15	yy/mm	59/10
@D16	yyyy/mm	1959/10
@D17		Windows Control Panel setting for Short Date
@D18		Windows Control Panel setting for Long Date
Alternate separators		
@D1.	mm.dd.yy	Period separator
@D2-	mm-dd-yyyy	Dash separator
@D5_	dd mm yy	Underscore produces space separator
@D6'	dd,mm,yyyy	Grave accent produces comma separator

See Also:

Standard Date

FORMAT

DEFORMAT

Environment Files

Time Pictures

@Tn[s][B]

-
- @T** All time pictures begin with @T.
- n** Determines the time picture format. Time picture formats range from 1 through 8. A leading zero (0) indicates zero-filled hours.
- s** A separation character. By default, colon (:) characters appear between the hour, minute, and second components of certain time picture formats. The following s indicators provide an alternate separation character for these formats.
- . (period) Produces periods
 - ' (grave accent) Produces commas
 - (hyphen) Produces hyphens
 - _ (underscore) Produces spaces
- B** Specifies that the format displays as blank when the value is zero.

Times may be stored in a numeric variable (usually a LONG), a TIME field (for Btrieve compatibility), or in a STRING declared with a time picture. A time stored in a numeric variable is called a "Standard Time." The stored value is the number of hundredths of a second since midnight. The picture token converts the value to one of the eight time formats.

For those time pictures which contain string data, the actual strings are customizable in an Environment file (.ENV). See the Internationalization section for more information.

Example:

<u>Picture</u>	<u>Format</u>	<u>Result</u>
@T1	hh:mm	17:30
@T2	hhmm	1730
@T3	hh:mmXM	5:30PM
@T03	hh:mmXM	05:30PM
@T4	hh:mm:ss	17:30:00
@T5	hhmmss	173000
@T6	hh:mm:ssXM	5:30:00PM
@T7		Windows Control Panel setting for Short Time
@T8		Windows Control Panel setting for Long Time

Alternate separators

@T1.	hh.mm	Period separator
@T1-	hh-mm	Dash separator
@T3_	hh mmXM	Underscore produces space separator
@T4'	hh,mm,ss	Grave accent produces comma separator

See Also: Standard Time, FORMAT, DEFORMAT, Environment Files

Pattern Pictures**@P[<][#][x]P[B]**

@P	All pattern pictures begin with the @P delimiter and end with the P delimiter. The case of the delimiters must be the same.
<	Specifies an integer position that is blank for leading zeroes.
#	Specifies an integer position.
x	Represents optional display characters. These characters appear in the final result string.
P	All pattern pictures must end with P. If a lower case @p delimiter is used, the ending P delimiter must also be lower case.
B	Specifies that the format displays as blank when the value is zero.

Pattern pictures contain optional integer positions and optional edit characters. Any character other than < or # is considered an edit character which will appear in the formatted picture string. The @P and P delimiters are case sensitive. Therefore, an upper case "P" can be included as an edit character if the delimiters are both lower case "p" and vice versa.

Pattern pictures do not recognize decimal points, in order to permit the period to be used as an edit character. Therefore, the value formatted by a pattern picture should be an integer. If a floating point value is formatted by a pattern picture, only the integer portion of the number will appear in the result.

Example:

<u>Picture</u>	<u>Value</u>	<u>Result</u>
@P###-##-####P	215846377	215-84-6377
@P<#/#/#P	103159	10/31/59
@P(###)###-####P	3057854555	(305)785-4555
@P###/###-####P	7854555	000/785-4555
@p<#:##PMp	530	5:30PM
@P<#' <#"P	506	5' 6"
@P<#lb. <#oz.P	902	91b. 2oz.
@P4##A-#P	112	411A-2
@PA##.C#P	312.45	A31.C2

Key-in Template Pictures

@K[@][#][<][x][N][?][^][_][|][K][B]

@K	All key-in template pictures begin with the @K delimiter and end with the K delimiter. The case of the delimiters must be the same.
@	Specifies only uppercase and lowercase alphabetic characters.
#	Specifies an integer 0 through 9.
<	Specifies an integer that is blank for high order zeros.
x	Represents optional constant display characters (any displayable character). These characters appear in the final result string.
\	Indicates the following character is a display character. This allows you to include any of the picture formatting characters (@,#,<,\,?,^,_,) within the string as a display character.
?	Specifies any character may be placed in this position.
^	Specifies only uppercase alphabetic characters in this position.
_	Underscore specifies only lowercase alphabetic characters in this position.
 	Allows the operator to "stop here" if there are no more characters to input. Only the data entered and any display characters up to that point will be in the string result.
K	All key-in template pictures must end with K. If a lower case @k delimiter is used, the ending K delimiter must also be lower case.
B	Specifies that the format displays as blank when the value is zero.

Key-in pictures may contain integer positions (# <), alphabet character positions (@ ^ _), any character positions (?), and display characters. Any character other than a formatting indicator is considered a display character, which appears in the formatted picture string. The @K and K delimiters are case sensitive. Therefore, an upper case "K" may be included as a display character if the delimiters are both lower case "k" and vice versa.

Key-in pictures are used specifically with STRING, PSTRING, and CSTRING fields to allow custom field editing control and validation. Using a key-in picture containing any of the alphabet indicators (@ ^ _) on a numeric entry field produces unpredictable results.

Using the Insert typing mode for a key-in picture could produce unpredictable results. Therefore, key-in pictures always receive data entry in Overwrite mode, even if the INS attribute is present.

Example:

<u>Picture</u>	<u>Value Entered</u>	<u>Result String</u>
@K###-##-####K	215846377	215-84-6377
@K##### #####K	33064	33064
@K##### #####K	330643597	33064-3597
@K<# ^^^ ##K	10AUG59	10 AUG 59
@K(###)@@-##\@##K	305abc4555	(305)abc-45@55
@K###/?##-####K	7854555	000/785-4555
@k<#:##^Mk	530P	5:30PM
@K<#' <#"K	506	5' 6"
@K4#_#A-#K	1g12	41g1A-2

4 - Entity Declarations

Complex Data Structures

GROUP (compound data structure)

```
label  GROUP( [ group ] ) [,PRE( )] [,DIM( )] [,OVER( )] [,NAME( )] [,EXTERNAL] [,DLL] [,STATIC]
        [,THREAD] [,BINDABLE] [, TYPE] [,PRIVATE] [,PROTECTED]
        declarations
        END
```

GROUP	A compound data structure.
<i>group</i>	The label of a previously declared GROUP or QUEUE structure from which it will inherit its structure. This may be a GROUP or QUEUE with the TYPE attribute.
PRE	Declare a label prefix for variables within the structure. Not valid on a GROUP within a FILE structure.
DIM	Dimension the variables into an array.
OVER	Share a memory location with another variable or structure.
NAME	Specify an alternate, "external" name for the field.
EXTERNAL	Specify the variable is defined, and its memory is allocated, in an external library. Not valid within FILE, QUEUE, or GROUP declarations.
DLL	Specify the variable is defined in a .DLL. This is required in addition to the EXTERNAL attribute.
STATIC	Specify the variable's memory is permanently allocated.
THREAD	Specify memory for the variable is allocated once for each execution thread. Also implicitly adds the STATIC attribute on Procedure Local data.
BINDABLE	Specify all variables in the group may be used in dynamic expressions.
TYPE	Specify the GROUP is a type definition for GROUPs passed as parameters.
PRIVATE	Specify the GROUP and all the component fields of the GROUP are not visible outside the module containing the CLASS methods. Valid only in a CLASS.
PROTECTED	Specify the variable is not visible outside base CLASS and derived CLASS methods. Valid only in a CLASS.
<i>declarations</i>	Multiple consecutive variable declarations.

A **GROUP** structure allows multiple variable declarations to be referenced by a single label. It may be used to dimension a set of variables, or to assign or compare sets of variables in a single statement. In large complicated programs, a GROUP structure is helpful for keeping sets of related data organized. A GROUP must be terminated by a period or the END statement.

The structure of a GROUP declared with the *group* parameter begins with the same structure as the named *group*; the GROUP inherits the fields of the named *group*. The GROUP may also contain its own *declarations* that follow the inherited fields. If the group parameter names a QUEUE or RECORD structure, only the fields are inherited and not the functionality implied by the QUEUE or RECORD.

When referenced in a statement or expression, a GROUP is treated as a STRING composed of all the variables within the structure. A GROUP structure may be nested within another data structure, such as a RECORD or another GROUP.

Because of their internal storage format, numeric variables (other than DECIMAL) declared in a group do not collate properly when treated as strings. For this reason, building a KEY on a GROUP that contains numeric variables may produce an unexpected collating sequence.

A GROUP with the BINDABLE attribute makes all the variables within the GROUP available for use in a dynamic expression. The contents of each variable's NAME attribute is the logical name used in the dynamic expression. If no NAME attribute is present, the label of the variable (including prefix) is used. Space is allocated in the .EXE for the names of all of the variables in the structure. This creates a larger program that uses more memory than it normally would. Therefore, the BINDABLE attribute should only be used when a large proportion of the constituent fields are going to be used.

A GROUP with the TYPE attribute is not allocated any memory; it is only a type definition for GROUPs that are passed as parameters to PROCEDURES. This allows the receiving procedure to directly address component fields in the passed GROUP. The parameter declaration on the PROCEDURE statement can instantiate a local prefix for the passed GROUP as it names the passed GROUP for the procedure, however this is not necessary if you use the Field Qualification syntax instead of prefixes. For example, PROCEDURE(LOC:PassedGroup) declares the procedure uses the LOC: prefix (along with the individual field names used in the type definition) to directly address component fields of the GROUP passed as the parameter.

The data elements of a GROUP with the DIM attribute (a structured array) are referenced using standard Field Qualification syntax with each subscript specified at the GROUP level at which it is dimensioned.

The WHAT and WHERE procedures allow access to the fields by their relative position within the GROUP structure.

Example:

```

PROGRAM
PassGroup  GROUP,TYPE           !Type-definition for passed GROUP parameters
F1         STRING(20)           ! first field
F2         STRING(1)            ! middle field
F3         STRING(20)           ! last field
        END

MAP
  MyProc1(PassGroup)           !Passes a GROUP defined the same as PassGroup
END

NameGroup  GROUP
First     STRING(20)           ! first name
Middle    STRING(1)            ! middle initial
Last      STRING(20)           ! last name
        END                     !End group declaration

NameGroup2 GROUP(PassGroup)    !Group that inherits PassGroup's fields
        ! resulting in NameGroup2.F1, NameGroup2.F2,
        ! and NameGroup2.F3
        END                     ! fields declared in this group

DateTimeGrp GROUP,DIM(10)      !Date/time array
Date      LONG                 ! Referenced as DateTimeGrp[1].Date
StartStopTime LONG,DIM(2)      ! Referenced as DateTimeGrp[1].Time[1]
        END                     !End group declaration

FileNames  GROUP,BINDABLE      !Bindable group
FileName   STRING(8),NAME('FILE') !Dynamic name: FILE
Dot        STRING('.')         !Dynamic name: Dot
Extension  STRING(3),NAME('EXT') !Dynamic name: EXT
        END

CODE
  MyProc1(NameGroup)           !Call proc passing NameGroup as parameter
  MyProc1(NameGroup2)         !Call proc passing NameGroup2 as parameter

MyProc1  PROCEDURE(PassedGroup) !Proc to receive GROUP parameter
LocalVar STRING(20)
CODE
  LocalVar = PassedGroup.F1     !Assign value in the first field to LocalVar
        !from passed parameter

```

See Also:

Field Qualification, WHAT, WHERE

CLASS (object declaration)

```

label  CLASS( [ parentclass ] ) [,EXTERNAL] [,IMPLEMENTS] [,DLL] [,STATIC] [,THREAD]
[,BINDABLE] [,MODULE( )]
           [, LINK( )] [, TYPE]
           [ data members and methods ]
END

```

CLASS	An object containing <i>data members</i> and <i>methods</i> that operate on the data.
<i>parentclass</i>	The label of a previously declared CLASS structure whose data and methods the new CLASS inherits. This may be a CLASS with the TYPE attribute.
EXTERNAL	Specify the object is defined, and its memory is allocated, in an external library.
IMPLEMENTS	Specify an INTERFACE for the CLASS. This adds additional methods to the implementation of the CLASS.
DLL	Specify the object is defined in a .DLL. This is required in addition to the EXTERNAL attribute.
STATIC	Specify the <i>data members'</i> memory is permanently allocated.
THREAD	Specify memory for the variables is allocated once for each execution thread. Also implicitly adds the STATIC attribute on Procedure Local data. Not valid with TYPE.
BINDABLE	Specify all variables in the class may be used in dynamic expressions.
MODULE	Specify the source code module containing the CLASS's member PROCEDURE definitions. This serves the same function as the MODULE structure within a MAP structure. If omitted, the member PROCEDURE definitions must all be in the same source code module containing the CLASS declaration.
LINK	Specify the source code module containing the CLASS's member PROCEDURE definitions is automatically added to the compiler's link list. This eliminates the need to specifically add the file to the project.
TYPE	Specify the CLASS is only a type definition and not also an object instance of the CLASS.

data members and methods

Data declarations and PROCEDURE prototypes. The *data members* may only be data declarations appropriate to a GROUP structure, and may include references to the same class (recursive classes). The WHAT and WHERE procedures allow access to the *data members* by their relative position within the CLASS structure.

A **CLASS** structure declares an object which contains *data members* (properties) and the *methods* (PROCEDURES) that act on that data. A CLASS structure must be terminated by a period or the END statement.

Derived CLASSES (Inheritance)

A CLASS declared with the *parentclass* parameter creates a *derived class* which inherits all the *data members and methods* of the named *parentclass*. The *derived class* may also contain its own *data members and methods*.

All *data members* explicitly declared in the *derived class* create new variables, and cannot be declared with the same labels as *data members* in the *parentclass*.

Any *method* prototyped in the *derived class* with the same name as a *method* in the *parentclass* overrides the inherited *method* if both have the same parameter lists. If the two *methods* have different parameter lists, they create polymorphic functions in the *derived class* that must obey the rules of Procedure Overloading.

Object Properties (Encapsulation)

Each instance of a CLASS, whether a base class, derived class, or a declared instance of either, contains its own set of *data members* (properties) specific to that instance. These may be private or public. However, there is only one copy of any inherited *methods* (residing in the CLASS that declared it) which any instance of that CLASS, or any of its *derived classes*, calls.

The *methods* of a CLASS with the TYPE attribute cannot be directly called (as *ClassName.Method*)--they must be called only as a member *methods* of the objects declared as the type (as *Object.Method*).

VIRTUAL Methods (Polymorphism)

If there is a *method* prototyped in the CLASS with the same label as a *method* in the *parentclass* with the VIRTUAL attribute, it must also be prototyped with the VIRTUAL attribute in the *derived class*.

The VIRTUAL attribute on both prototypes creates virtual methods that allow the *methods* in a *parentclass* to call the same named VIRTUAL *methods* in the *derived class* to perform functions specific to the *derived class* that the *parentclass* does not know about.

VIRTUAL *methods* in the *derived class* may directly call the *parentclass* method of the same name by prepending PARENT to the method's name. This allows incremental derivation wherein a *derived class* method may simply call down to the *parentclass* method to perform its functionality, then extend it for the requirements of the *derived class*.

Scoping Issues

The scope of an object is dependent upon where it is declared. Generally, a declared object comes into scope at the CODE statement following its declaration and goes out of scope at the end of the related executable code section. A dynamically instantiated object (using NEW) shares the scope of the executable code section in which it is instantiated.

An Object declared:

- As Global data is in scope throughout the application.
- As Module data is in scope throughout the module.
- As Local data is in scope only in the procedure, except ...

Methods prototyped in a derived CLASS declaration within a procedure's Local data section are Local Derived Methods and share the declaring procedure's scope for all local data declarations and routines. The methods must be defined within the same source module as the procedure within which the CLASS is declared and must immediately follow the procedure within that source--that is, they 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.

For example:

```

SomeProc    PROCEDURE
MyLocalVar  LONG
MyDerivedClass CLASS(MyClass) !Derived class with a virtual method
MyProc      PROCEDURE,VIRTUAL
            END

CODE
!SomeProc main executable code goes here
!SomeProc ROUTINEs goes here
MyRoutine  ROUTINE
!Routine code goes here

!MyDerivedClass methods immediately follow:

MyDerivedClass.MyProc PROCEDURE
CODE
MyLocalVar = 10           !MyLocalVar is still in scope, and available for u
DO MyRoutine             !MyRoutine is still in scope, and available for u

!Any other procedures in the same module go here, following all
! derivd class methods

```

Instantiation

You declare an instance of a CLASS (an object) by simply naming the CLASS as the data type of the new instance, or by executing the NEW procedure in a reference assignment statement to a reference variable for that named CLASS. Either way, the new instance inherits all the *data members and methods* of the CLASS for which it is an instance. All the attributes of a CLASS except MODULE and TYPE are valid on an instance declaration.

If there is no TYPE attribute on the CLASS, the CLASS structure itself declares both the CLASS and an object instance of that CLASS. A CLASS with the TYPE attribute does not create an object instance of the CLASS.

For example, the following CLASS declaration declares the CLASS as a data type and an object of that type:

```
MyClass CLASS           !Both a data type declaration and an object instance
MyField  LONG
MyProc   PROCEDURE
END
```

while this only declares the CLASS as a data type:

```
MyClass CLASS,TYPE !Only a data type declaration
MyField  LONG
MyProc   PROCEDURE
END
```

It is preferable to directly declare object instances as the CLASS data type rather than as a reference to the CLASS. This results in smaller quicker code and does not require you to use NEW and DISPOSE to explicitly create and destroy the object instance. The advantage of using NEW and DISPOSE is explicit control over the lifetime of the object. For example:

```
MyClass CLASS,TYPE
MyField  LONG
MyProc   PROCEDURE
END

OneClass MyClass           !Declared object instance, smaller and quicker
TwoClass &MyClass         !Object reference, must use New and DISPOSE
CODE
!execute some code here
TwoClass &= NEW(MyClass) !The lifetime of the object starts here
!execute some code here
DISPOSE(TwoClass)        ! and extends only to here
!execute some code here
```

Another advantage of declaring the object is the ability to declare the object with any of the attributes available for the CLASS declaration itself (except TYPE and MODULE). For instance, you can declare an object with the THREAD attribute, whether the CLASS is declared with THREAD or not.

Note:

The constructors and destructors for threaded classes are called for every thread. Every new thread gets new instances of CLASSES and variables declared at the global level with the THREAD attribute. The RTL calls constructors for the threaded classes when the thread is started and the destructors when the thread is ended. In previous Clarion versions they were called only when the main thread started and ended.

The lifetime of an object depends on how it is instantiated:

- An object declared in the Global data section or a Module's data section is instantiated at the CODE statement following the PROGRAM statement and de-instantiated when the application terminates.
- A reference to an object is instantiated by the NEW statement, and de-instantiated by the DISPOSE statement.
- An object declared in a procedure's Local data section is instantiated at the CODE statement following the PROCEDURE statement and de-instantiated when a RETURN (implicit or explicit) executes to terminate the procedure.

Data (Property) Initialization

The simple data type *data members* of an object are automatically allocated memory and initialized to blank or zero (unless the AUTO attribute is specified) when the object comes into scope. The allocated memory is returned to the operating system when the object goes out of scope.

The reference variable *data members* of an object are not allocated memory and are not initialized when the object comes into scope--you must specifically execute a reference assignment or a NEW statement. These references variables are not automatically cleared when the object goes out of scope, so you must DISPOSE of all NEWed properties before the object goes out of scope.

Constructors and Destructors

A CLASS *method* labelled "Construct" is a constructor method which is automatically invoked when the object comes into scope, immediately after the *data members* of the object are allocated and initialized. The "Construct" *method* may not receive any parameters and may not be VIRTUAL. You may explicitly call the "Construct" method in addition to its automatic invocation.

If an object is an instance of a derived CLASS and both the *parentclass* and the derived CLASS contain constructors and the derived CLASS's constructor does not have the REPLACE attribute, then the *parentclass* constructor is automatically invoked at the beginning of the derived CLASS's constructor. If the derived CLASS's constructor does have the REPLACE attribute, then only derived CLASS's constructor is automatically invoked (the derived CLASS's constructor method can explicitly call PARENT.Construct if it needs to).

A CLASS *method* labelled "Destruct" is a destructor method which is automatically invoked when the object leaves scope, immediately before the *data members* of the object are de-allocated. The "Destruct" *method* may not receive any parameters. You may explicitly call the "Destruct" method in addition to its automatic invocation.

If an object is an instance of a derived CLASS and both the *parentclass* and the derived CLASS contain destructors and the derived CLASS's destructor does not have the REPLACE attribute, then the *parentclass* destructor is automatically invoked at the end of the derived CLASS's destructor. If the derived CLASS's destructor does have the REPLACE attribute, then only derived CLASS's destructor is automatically invoked (the derived CLASS's destructor method can explicitly call PARENT.Destruct if it needs to).

Public, PRIVATE, and PROTECTED (Encapsulation)

Public *data members and methods* of a CLASS or derived CLASS are declared without either the PRIVATE or PROTECTED attributes. Public *data members and methods* are visible to all the *methods* of the declaring CLASS, and derived CLASSES, and any code where the object is in scope.

Private *data members and methods* are declared with the PRIVATE attribute. Private *data members and methods* are visible only to the *methods* of the CLASS within which they are declared and any other procedures contained in the same source code module.

Protected *data members and methods* are declared with the PROTECTED attribute. Protected *data members and methods* are visible only to the *methods* of the CLASS within which they are declared, and to the *methods* of any CLASS derived from the CLASS within which they are declared.

Method Definition

The PROCEDURE definition of a *method* (its executable code, not its prototype) is external to the CLASS structure. The *method's* definition must either prepend the label of the CLASS to the label of the PROCEDURE, or name the CLASS (and label it SELF) as the first (implicit) parameter in the list of parameters passed in to the PROCEDURE.

Remember that on the PROCEDURE definition statement you are assigning labels for use within the method to all the passed parameters, and so, since the CLASS's label is the data type of the implicit first parameter, you must use SELF as the assigned label for the CLASS name parameter. For example, for the following CLASS declaration:

```
MyClass CLASS
MyProc  PROCEDURE(LONG PassedVar)      !The method takes 1 parameter
      END
```

you may define the MyProc PROCEDURE either as:

```
MyClass.MyProc PROCEDURE(LONG PassedVar)  !Prepend the CLASS name to
      CODE                                !the method's label
```

or as:

```
MyProc PROCEDURE(MyClass SELF, LONG PassedVar) !The CLASS name is the
      CODE
! implicit first parameter's data type, labeled SELF
```

Referencing an Object's properties and methods in your code

You must reference the *data members* of a CLASS using Clarion's Field Qualification syntax. To do this, you prepend the label of the CLASS (if it is an object instance of itself) or the label of an object instance of the CLASS to the label of the *data member*.

For example, for the following CLASS declarations:

```
MyClass CLASS          !Without TYPE, this is also an object instance
MyField  LONG          ! in addition to a class type declaration
MyProc   PROCEDURE
      END
MyClass2 MyClass      !Declare another object instance of MyClass
```

you must reference the two MyField variables from procedures external to the object as:

```
MyClass.MyField = 10  !References the MyClass CLASS declaration's object
MyClass2.MyField = 10 !References the MyClass2 declaration's object
```

You may call the *methods* of a CLASS either using Field Qualification syntax (by prepending the label of the CLASS to the label of the *method*), or by passing the label of the CLASS as the first (implicit) parameter in the list of parameters passed to the PROCEDURE.

For example, for the following CLASS declaration:

```
MyClass CLASS
MyProc  PROCEDURE
      END
```

you may call the MyProc PROCEDURE either as:

```
CODE
MyClass.MyProc
```

or as:

```
CODE
MyProc(MyClass)
```

SELF and PARENT

Within the *methods* of a CLASS, the *data members and methods* of the current object's instance are referenced with SELF prepended to their labels instead of the name of the CLASS. This allows the *methods* to generically reference the *data members and methods* of the currently executing instance of the CLASS, without regard to whether it is executing the *parentclass*, a *derived class*, or any instance of either. This is also the mechanism that allows a *parentclass* to call virtual *methods* of a *derived class*.

For example, expanding on the previous example, MyField is referenced within the MyClass.MyProc method as:

```
MyClass.MyProc PROCEDURE
CODE
SELF.MyField = 10    !Assign to the current object instance's property
```

The *data members and methods* of a *parentclass* can be directly referenced from within the methods of a *derived class* with PARENT prepended to their labels instead of SELF.

For example:

```
MyDerivedClass.MyProc PROCEDURE
CODE
!execute some code
PARENT.MyProc        !Call the base class method
!execute some more code
!
```

Additional Example:

```

!The ClassPrg.CLW file contains:
PROGRAM
MAP.                                !MAP required to get BUILTINS.CLW

OneClass CLASS                       !Base class
NameGroup GROUP                     !Reference as OneClass.NameGroup
First  STRING(20)                   !reference as OneClass.NameGroup.First
Last   STRING(20)                   !reference as OneClass.NameGroup.Last
END

BaseProc PROCEDURE(REAL Parm)        !Declare method prototype
Func     PROCEDURE(REAL Parm),STRING,VIRTUAL !Declare virtual method prototype
Proc     PROCEDURE(REAL Parm),VIRTUAL !Declare virtual method prototype
END                                       !End CLASS declaration

TwoClass CLASS(OneClass),MODULE('TwoClass.CLW') !Derived from OneClass
Func     PROCEDURE(LONG Parm),STRING !replaces OneClass.Func
Proc     PROCEDURE(STRING Msg, LONG Parm) !Functionally overloaded
END

ClassThree CLASS(TwoClass),MODULE('Class3.CLW') !Derived from TwoClass
Func     PROCEDURE(<STRING Msg>,LONG Parm),STRING,VIRTUAL
Proc     PROCEDURE(REAL Parm),VIRTUAL
END

ClassFour ClassThree                !Declare an instance of ClassThree
ClassFive ClassThree                !Declare an instance of ClassThree

CODE
OneClass.NameGroup = '|OneClass Method' !Assign values to each instance of NameGroup
TwoClass.NameGroup = '|TwoClass Method'
ClassThree.NameGroup = '|ClassThree Method'
ClassFour.NameGroup = '|ClassFour Method'
MESSAGE(OneClass.NameGroup & OneClass.Func(1.0))!Calls OneClass.Func
MESSAGE(TwoClass.NameGroup & TwoClass.Func(2)) !Calls TwoClass.Func

MESSAGE(ClassThree.NameGroup & ClassThree.Func('|Call ClassThree.Func',3.0))
!Calls ClassThree.Func

MESSAGE(ClassFour.NameGroup & ClassFour.Func('|Call ClassFour.Func',4.0))
!Also Calls ClassThree.Func

OneClass.BaseProc(5)                !BaseProc Calls OneClass.Proc & Func
BaseProc(TwoClass,6)                !BaseProc Also calls OneClass.Proc & Func
TwoClass.Proc('Second Class',7)    !Calls TwoClass.Proc (overloaded)
ClassThree.BaseProc(8)              !BaseProc Calls ClassThree.Proc & Func
ClassFour.BaseProc(9)               !BaseProc Also Calls ClassThree.Proc & Func

```

```

Proc(ClassFour,'Fourth Class',10)           !Calls TwoClass.Proc (overloaded)

OneClass.BaseProc  PROCEDURE(REAL Parm)    !Definition of OneClass.BaseProc
CODE
MESSAGE(Parm & SELF.NameGroup & '|BaseProc executing|calling SELF.Proc Virtual method')
SELF.Proc(Parm)                               !Calls virtual method
MESSAGE(Parm & SELF.NameGroup & '|BaseProc executing|calling SELF.Func Virtual method')
MESSAGE(SELF.NameGroup & SELF.Func(Parm)) !Calls virtual method

OneClass.Func     PROCEDURE(REAL Parm)      !Definition of OneClass.Func
CODE
RETURN('|Executing OneClass.Func - ' & Parm)

Proc              PROCEDURE(OneClass SELF,REAL Parm)  !Definition of OneClass.Proc
CODE
MESSAGE(SELF.NameGroup & ' |Executing OneClass.Proc - ' & Parm)

!The TwoClass.CLW file contains:
MEMBER('ClassPrg')

Func             PROCEDURE(TwoClass SELF,LONG Parm)   !Definition of TwoClass.Func
CODE
RETURN('|Executing TwoClass.Func - ' & Parm)

TwoClass.Proc   PROCEDURE(STRING Msg,LONG Parm)      !Definition of TwoClass.Proc
CODE
MESSAGE(Msg & ' |Executing TwoClass.Proc - ' & Parm)

!The Class3.CLW file contains:
MEMBER('ClassPrg')

ClassThree.Func PROCEDURE(<STRING Msg>,LONG Parm) !Definition of ClassThree.Func
CODE
SELF.Proc(Msg,Parm)                               !Call TwoClass.Proc (overloaded)
RETURN(Msg & ' |Executing ClassThree.Func - ' & Parm)

ClassThree.Proc PROCEDURE(REAL Parm)                !Definition of ClassThree.Proc
CODE
SELF.Proc('Called from ClassThree.Proc',Parm)     !Call TwoClass.Proc
MESSAGE(SELF.NameGroup & ' |Executing ClassThree.Proc - ' & Parm)

```

See Also: Field Qualification, MODULE, PROCEDURE Prototypes, Procedure Overloading, WHAT, WHERE

INTERFACE (class behavior definition)

```
label      INTERFACE ( [ parentinterface ] ) [, TYPE]
           [ methods ]
           END
```

INTERFACE A collection of methods to be used by the class that implements the interface.

parentinterface The label of a previously declared INTERFACE structure whose methods are inherited by the new INTERFACE. This may be an INTERFACE with the TYPE attribute.

TYPE Specify the INTERFACE is only a type definition. TYPE is implicit on an INTERFACE but may be explicitly specified.

methods PROCEDURE prototypes

An **INTERFACE** is a structure, which contains the *methods* (PROCEDURES) that define the behavior to be implemented by a CLASS. It cannot contain any property declarations. All methods defined within the INTERFACE are implicitly virtual. A period or the END statement must terminate an INTERFACE structure.

Derived INTERFACES (Inheritance)

An INTERFACE declared with the *parentinterface* parameter creates a *derived interface* that inherits all the *methods* of the named *parentinterface*. The *derived interface* may also contain its own *methods*.

Any *method* prototyped in the *derived interface* with the same name as a *method* in the *parentinterface* overrides the inherited *method* if both have the same parameter lists. If the two *methods* have different parameter lists, they create polymorphic functions in the *derived interface* that must follow the rules of Procedure Overloading.

VIRTUAL Methods (Polymorphism)

All methods in an INTERFACE are implicitly virtual, although the virtual attribute may be explicitly specified for clarity.

VIRTUAL *methods* in the *derived interface* may directly call the *parentinterface* method of the same name by prepending PARENT to the method's name. This allows incremental derivation wherein a *derived interface* method may simply call up to the *parentinterface* method to perform its functionality, and then extend it for the requirements of the *derived interface*.

Method Definition

The PROCEDURE definition of a *method* (its executable code, not its prototype) is defined by the CLASS that is implementing the INTERFACE. All methods for an interface must be defined in the IMPLEMENTING class.

Referencing INTERFACE methods in your code

You must call the *methods* of an INTERFACE by using dot notation syntax (by prepending the label of the CLASS to the label of the INTERFACE to the label of the *method*).

For example, using the following INTERFACE and CLASS declaration:

```
MyInterface INTERFACE
MyProc      PROCEDURE
            END
MyClass     CLASS, IMPLEMENTS(MyInterface)
            END
```

You may call the MyProc PROCEDURE as:

```
CODE
MyClass.MyInterface.MyProc
```

See Also:

IMPLEMENTS

File Structures

FILE (declare a data file structure)

```
label  FILE,DRIVER( ) [,CREATE] [,RECLAIM] [,OWNER( )] [,ENCRYPT] [,NAME()] [,PRE( )]
                                     [,BINDABLE] [,THREAD] [,EXTERNAL] [,DLL] [,OEM]
```

```
label  [INDEX( )]
```

```
label  [KEY( )]
```

```
label  [MEMO( )]
```

```
label  [BLOB]
```

```
[label] RECORD
```

```
[label]  fields
```

```
END
```

```
END
```

label A valid Clarion label for the FILE, INDEX, KEY, MEMO, BLOB, RECORD, or *field* (PROP:Label).

FILE Declares a data file.

DRIVER Specifies the data file type (PROP:DRIVER). The DRIVER attribute is required on all FILE structure declarations.

CREATE Allows the file to be created with the CREATE statement during program execution (PROP:CREATE).

RECLAIM Specifies reuse of deleted record space (PROP:RECLAIM).

OWNER Specifies the password for data encryption (PROP:OWNER).

ENCRYPT Encrypt the data file (PROP:ENCRYPT).

NAME Set DOS filename specification (PROP:NAME).

PRE Declare a label prefix for the structure.

BINDABLE Specify all variables in the RECORD structure may be used in dynamic expressions.

THREAD Specify memory for the record buffer is separately allocated for each execution thread, when the file is opened on the thread (PROP:THREAD).

EXTERNAL	Specify the FILE is defined, and the memory for its record buffer is allocated, in an external library.
DLL	Specify the FILE is defined in a .DLL. This is required in addition to the EXTERNAL attribute.
OEM	Specify string data is converted from OEM ASCII to ANSI when read from disk and ANSI to OEM ASCII before writing to disk (PROP:OEM).
INDEX	Declare a static file access index which must be built at run time.
KEY	Declare a dynamically updated file access index.
MEMO	Declare a variable length text field up to 64K in length.
BLOB	Declare a variable length memo field which may be greater than 64K in length.
RECORD	Declare a record structure for the <i>fields</i> . A RECORD structure is required in all FILE structure declarations.

fields Data elements in the RECORD structure.

FILE declares a data file structure which is an exact description of a data file residing on disk. The label of the FILE structure is used in file processing statements and procedures to effect operations on the disk file. The FILE structure must be terminated by a period or the END statement.

All attributes of the FILE, KEY, INDEX, MEMO, data declaration statements, and the data types which a FILE may contain, are dependent upon the support of the file driver. Anything in the FILE declaration which is not supported by the file system specified in the DRIVER attribute will cause a file driver error when the FILE is opened. Attribute and/or data type exclusions for a specific file system are listed in each file driver's documentation.

At run-time, the RECORD structure is assigned memory for a data buffer where records from the disk file may be processed by executable statements. This record buffer is always allocated static memory on the heap, even if the FILE is declared in a local data section. A RECORD structure is required in a FILE structure. Memory for a data buffer for any MEMO fields is allocated only when the FILE is opened, and de-allocated when the FILE is closed. The memory for BLOB fields is allocated as needed once the FILE is open.

A FILE with the BINDABLE attribute declares all the variables within the RECORD structure as available for use in a dynamic expression, without requiring a separate BIND statement for each (allowing BIND(file) to enable all the fields in the file). The contents of each variable's NAME attribute is the logical name used in the dynamic expression. If no NAME attribute is present, the label of the variable (including any prefix) is used. Space is allocated in the .EXE for the names of all of the variables in the structure. This creates a larger program that uses more memory than it normally would. Therefore, the BINDABLE attribute should only be used when a large proportion of the constituent fields are going to be used.

A FILE with the THREAD attribute declares a separate record buffer (and file control block) for each execution thread that OPENS the FILE. If the thread does not OPEN the file, no record buffer is allocated for the file on that thread. If a NAME attribute is defined for the file and declared as "STRING, STATIC", it will need to explicitly declare the THREAD attribute if a different file name will be opened on each thread (or PROP:Name may be used to specify the file name).

Note:

Any FILE declared in the local scope of a PROCEDURE or ROUTINE is treated as threaded, regardless of the presence of the THREAD attribute in its declaration.

A FILE with the EXTERNAL attribute is declared and may be referenced in Clarion code, but is not allocated memory. The memory for the FILE's record buffer is allocated by the external library. This allows a Clarion program access to FILEs declared as public in external libraries.

Related Procedures:

BUFFER, BUILD, CLOSE, COPY, CREATE, EMPTY, FLUSH, LOCK, NAME, OPEN, PACK, RECORDS, REMOVE, RENAME, SEND, SHARE, STATUS, STREAM, UNLOCK, ADD, APPEND, BOF, BYTES, DELETE, DUPLICATE, EOF, GET, HOLD, NEXT, NOMEMO, POINTER, POSITION, PREVIOUS, PUT, RELEASE, REGET, RESET, SET, SKIP, WATCH

Example:

```
Names FILE,DRIVER('Clarion') !Declare a file structure
Rec   RECORD                 !Required record structure
Name  STRING(20)             !containing one or more data elements
      END
      END                     !End file and record declaration
```

See Also:

KEY

INDEX

MEMO

BLOB

RECORD

INDEX (declare static file access index)

label **INDEX**([-/+][*field*],...[-/+][*field*]) [,**NAME**()] [,**NOCASE**] [,**OPT**]

<i>label</i>	The label of the INDEX (PROP:Label).
INDEX	Declares a static index into the data file.
-/+	The - (<i>minus sign</i>) preceding an index component <i>field</i> specifies descending order for that component. If omitted, or + (<i>plus sign</i>) the component is sorted in ascending order.
<i>field</i>	The label of a field in the RECORD structure of the FILE in which the INDEX is declared. The <i>field</i> is an index component. Fields declared with the DIM attribute (arrays) may not be used as index components.
NAME	Specifies the disk file specification for the INDEX (PROP:NAME).
OPT	Excludes those records with null values (zero or blank) in all index component fields (PROP:OPT).
NOCASE	Specifies case insensitive sort order (PROP:NOCASE).

INDEX declares a "static key" for a FILE structure. An INDEX is updated only by the BUILD statement. It is used to access records in a different logical order than the "physical order" of the file. An INDEX may be used for either sequential file processing or direct random access.

An INDEX always allows duplicate entries. An INDEX may have more than one component *field*. The order of the components determines the sort sequence of the index. The first component is the most general, and the last component is the most specific. Generally, a data file may have up to 255 indexes (and/or keys) and each index may be up to 255 bytes, but the exact numbers are file driver dependent.

An INDEX declared without a *field* creates a "dynamic index." A dynamic index may use any field (or fields) in the RECORD as components (except arrays). The component fields of a dynamic index are defined at run time in the second parameter of the BUILD statement. The same dynamic index declaration may be built and re-built using different components each time.

Example:

```
Names    FILE,DRIVER( 'TopSpeed' ),PRE(Nam)
NameNdx    INDEX(Nam:Name),NOCASE    !Declare the name index
NbrNdx    INDEX(Nam:Number),OPT    !Declare the number index
DynamicNdx INDEX( )    !Declare a dynamic index
Rec        RECORD
Name        STRING( 20)
Number      SHORT
            END
            END
```

See Also: SET, GET, KEY, BUILD

KEY (declare dynamic file access index)

label **KEY**([-/+]*field*,...[-/+]*field*) [,DUP] [,NAME()] [,NOCASE] [,OPT] [,PRIMARY]

<i>label</i>	The label of the KEY (PROP:Label).
KEY	Declares a dynamically maintained index into the data file.
-/+	The - (<i>minus sign</i>) preceding a key component <i>field</i> specifies descending order for that component. If omitted, or + (<i>plus sign</i>), the component is sorted in ascending order.
<i>field</i>	The label of a field in the RECORD structure of the FILE in which the KEY is declared. The <i>field</i> is a key component. A field declared with the DIM attribute (an array) may not be used as a key component.
NAME	Specifies the disk file specification of the KEY (PROP:NAME).
DUP	Allows multiple records with duplicate values in their key component fields (PROP:DUP).
NOCASE	Specifies case insensitive sort order (PROP:NOCASE).
OPT	Excludes, from the KEY, those records with null (zero or blank) values in all key component fields (PROP:OPT).
PRIMARY	Specifies the KEY is the file's relational primary key (a unique key containing all records in the file) (PROP:PRIMARY).

A **KEY** is an index into the data file which is automatically updated whenever records are added, changed, or deleted. It is used to access records in a different logical order than the "physical order" of the file. A **KEY** may be used for either sequential file processing or direct random access.

A **KEY** may have more than one component *field*. The order of the components determines the sort sequence of the key. The first component is the most general, and the last component is the most specific. Generally, a data file may have up to 255 keys (and indexes) and each key may be up to 255 bytes, but the exact numbers are file driver dependent.

Example:

```
Names  FILE,DRIVER('Clarion'),PRE(Nam)
NameKey KEY(Nam:Name),NOCASE,DUP      !Declare the name key
NbrKey  KEY(Nam:Number),OPT          !Declare the number key
Rec     RECORD
Name    STRING(20)
Number  SHORT
      END
      END
CODE
Nam:Name = 'Clarion Software'      !Initialize key field
GET(Names,Nam:NameKey)            !Get the record
SET(Nam:NbrKey)                   !Set sequential by number
```

See Also:

SET, GET, INDEX, BUILD, PACK

MEMO (declare a text field)

label **MEMO**(*length*) [,**BINARY**] [,**NAME**()]

<i>label</i>	The label of the MEMO (PROP:Label).
MEMO	Declares a fixed-length string which is stored variable-length on disk per record.
<i>length</i>	A numeric constant that determines the maximum number of characters. The maximum range is unlimited in 32-bit applications (dependent on the file driver's MEMO support).
BINARY	Declares the MEMO a storage area for binary data (PROP:BINARY).
NAME	Specifies the disk filename for the MEMO field (PROP:NAME).

MEMO declares a fixed-length string field which is stored variable-length on disk. The *length* parameter defines the maximum size of a memo. A MEMO must be declared before the RECORD structure. Memory is allocated for a MEMO field's buffer when the file is opened, and is de-allocated when the file is closed. MEMO fields are usually displayed in TEXT fields in SCREEN and REPORT structures.

Generally, up to 255 MEMO fields may be declared in a FILE structure. The exact size and number of MEMO fields, and their manner of storage on disk, is file driver dependent.

Example:

```

Names    FILE,DRIVER('Clarion'),PRE(Nam)
NameKey   KEY(Nam:Name)
NbrKey    KEY(Nam:Number)
Notes    MEMO(4800)            !Memo, 4800 bytes
Rec        RECORD
Name       STRING(20)
Number     SHORT
             END
             END

```

BLOB (declare a variable-length field)

label **BLOB** [,**BINARY**] [,**NAME**()]

label The label of the BLOB (PROP:Label).

BLOB Declares a variable-length string stored on disk per record which may be greater than 64K

BINARY Declares the BLOB a storage area for binary data (PROP:BINARY).

NAME Specifies the disk filename for the BLOB field (PROP:NAME).

BLOB (Binary Large Object) declares a string field which is completely variable-length and may be greater than 64K in size. A BLOB must be declared before the RECORD structure. Generally, up to 255 BLOB fields may be declared in a FILE structure (the exact number and their manner of storage on disk is file driver dependent).

A BLOB may not be used as a variable--you may not name a BLOB as a control's USE attribute, or directly assign data to or from the BLOB. You can use PROP:Handle to get the Windows handle to the BLOB entity and assign one BLOB to another: get the handle of both BLOB entities and then assign one BLOB's handle to the other BLOB's handle. A BLOB may not be accessed "as a whole;" you must either use Clarion's string slicing syntax to access the data (unlimited in 32-bit), or PROP:ImageBlob. The individual bytes of data in the BLOB are numbered starting with zero (0), not one (1).

The SIZE procedure returns the number of bytes contained in the BLOB field for the current record in memory. You can also get (and set) the size of a BLOB using PROP:Size. You may set the size of the BLOB before assigning data to a new BLOB using string slicing, but it is not necessary as the size is automatically set by the string slice operation. You can also use PROP:ImageBlob to store and retrieve graphic images without first setting PROP:Size. It is a good idea to first set PROP:Size to zero (0) before assigning data to a BLOB that has not previously contained data, to eliminate any "junk" leftover from any previously accessed BLOB. When assigning from one BLOB to another using PROP:Handle, you may need to use PROP:Size to adjust the size of the destination BLOB to the size of the source BLOB. PROP:Touched can be used to determine if the contents of the BLOB has changed since it was retrieved from disk.

Example:

```

ArchiveFile PROCEDURE
Names FILE,DRIVER('TopSpeed')
NameKey KEY(Name)
Notes BLOB !Can be larger than 64K
Rec RECORD
Name STRING(20)
END
END

ArcNames FILE,DRIVER('TopSpeed')
Notes BLOB
Rec RECORD
Name STRING(20)
END
END

CODE
SET(Names)
LOOP
NEXT(Names)
IF ERRORCODE() THEN BREAK.
ArcNames.Rec = Names.Rec !Assign rec data to Archive
ArcNames.Notes{PROP:Handle} = Names.Notes{PROP:Handle} !Assign BLOB to Archive
IF ERRORCODE() = 80
MESSAGE('BLOB size is too large')
BREAK
END
ArcNames.Notes{PROP:Size} = Names.Notes{PROP:Size} ! and adjust the size
ADD(ArcNames)
END

StoreFileInBlob PROCEDURE !Stores any disk file into a B
DosFileName STRING(260),STATIC
LastRec LONG
SavPtr LONG(1) !Start at 1
FileSize LONG

DosFile FILE,DRIVER('DOS'),PRE(DOS),NAME(DosFileName)
Record RECORD
F1 STRING(2000)
END
END

BlobStorage FILE,DRIVER('TopSpeed'),PRE(STO)
File BLOB,BINARY
Record RECORD

```

```

FileName      STRING(64)
              END
              END

CODE
IF NOT FILEDIALOG('Choose File to Store',DosFileName,,0010b) THEN RETURN.
OPEN(BlobStorage)           !Open the BLOB file
STO:FileName = DosFileName   ! and store the filename
OPEN(DosFile)               !Open the file
FileSize = BYTES(DosFile)    !Get size of file
STO:File{PROP:Size} = FileSize ! and set the BLOB to store the file
LastRec = FileSize % SIZE(DOS:Record) !Check for short record at end of file
LOOP INT(FileSize/SIZE(DOS:Record)) TIMES
  GET(DosFile,SavPtr)        !Get each record
  ASSERT(NOT ERRORCODE())
  STO:File[SavPtr - 1 : SavPtr + SIZE(DOS:Record) - 2] = DOS:Record
                              !String slice data into BLOB
  SavPtr += SIZE(DOS:Record) !Compute next record pointer
END
IF LastRec                   !If short record at end of file
  GET(DosFile,SavPtr)        !Get last record
  ASSERT(BYTES(DosFile) = LastRec) ! size read should match computed size
  STO:File[SavPtr - 1 : SavPtr + LastRec - 2] = DOS:Record
END
ADD(BlobStorage)
ASSERT(NOT ERRORCODE())
CLOSE(DosFile);CLOSE(BlobStorage)

```

See Also: PROP:ImageBlob, PROP:Size , Implicit String Arrays and String Slicing, BLOBtoFILE

FILEtoBLOB

RECORD (declare record structure)

[label] **RECORD** [,PRE()] [,NAME()]

fields

END

RECORD Declares the beginning of the data structure within the FILE declaration.

fields Multiple variable declarations.

PRE Specify a label prefix for the structure.

NAME Specifies an external name for the RECORD structure.

The **RECORD** statement declares the beginning of the data structure within the FILE declaration. A RECORD structure is required in a FILE declaration. Each *field* is an element of the RECORD structure. The length of a RECORD structure is the sum of the length of its fields. When the label of a RECORD structure is used in an assignment statement, expression, or parameter list, it is treated as a GROUP data type.

At run time, static memory is allocated as a data buffer for the RECORD structure. The *fields* in the record buffer are available whether the file is open or closed.

If the *fields* contain variable declarations with initial values, that initial value is only used to determine the size of the variable, the record buffer is not initialized to the value. For example, a STRING('abc') field declaration creates a three-byte string, but its value is not automatically initialized to 'abc' unless the program's executable code assigns it that value.

Records from the data file on disk are read into the data buffer with the NEXT, PREVIOUS, GET, or REGET statements. Data in the *fields* are processed, then written to the data file as a single RECORD unit by the ADD, APPEND, PUT, or DELETE statements.

The WHAT and WHERE procedures allow access to the *fields* by their relative position within the RECORD structure.

Example:

```
Names  FILE,DRIVER('Clarion')  !Declare a file structure
Record RECORD                ! begin record declaration
Name    STRING(20)           ! declare name field
Number  SHORT                ! declare number field
      END
      END                    !End file, end record declaration
```

See Also: FILE, NEXT, PREVIOUS, GET, REGET, ADD, APPEND, PUT, DELETE, WHAT, WHERE

Null Data Processing

The concept of a null "value" in a field of a FILE 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.

In expressions, null does not equal blank or zero. Therefore, any expression which compares the value of a field from a FILE 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 is entirely dependent upon the file driver. Some file drivers support the null field concept (SQL drivers, for the most part), while others do not. Consult the documentation for the specific file driver to determine whether or not your file system's driver supports nulls.

See Also:

NULL

SETNULL

SETNONULL

FILE Structure Properties

Multi-File Properties

PROP:Label PROP:NAME PROP:Type

File Properties

PROP:CREATE	PROP:KEY	PROP:OWNER
PROP:Driver	PROP:Keys	PROP:RECLAIM
PROP:ENCRYPT	PROP:OEM	PROP:THREAD

Key Properties

PROP:Ascending	PROP:DUP	PROP:OPT
PROP:BINARY	PROP:FIELD	PROP:Over
PROP:Blobs	PROP:Fields	PROP:Places
PROP:Components	PROP:Memos	PROP:PRIMARY
PROP:Dim	PROP:NOCASE	PROP:Size

The following properties are all elements of a FILE data structure. They describe the attributes, fields, keys, memos, and blobs that may occur within a FILE structure. All these FILE structure properties are READ ONLY except: PROP:NAME (which can be used to change the name of a field in a file), PROP:OWNER, and PROP:DriverString. Assigning values to these properties overrides any values in the relevant declared attributes

Some properties are specific to the FILE and take the label of the FILE structure as the *target*, others are specific to a KEY (or INDEX) and take the label of the KEY (or INDEX) as the *target*, and others are specific to a BLOB and take the label of the BLOB as the *target*. Several properties are arrays, which take the number of the specific field or key as their element number to identify which field or key to return.

Each field that appears within the RECORD structure receives a positive number. In the RECORD structure, field declarations begin with 1 and increment by 1 for each subsequent field, in the order in which they appear within the RECORD structure. Terminating END statements for GROUP structures are not numbered, as they are not a field declaration.

MEMO fields are numbered negatively. MEMO declarations begin with -1 and decrement by 1 for each subsequent MEMO, in the order in which they appear within the FILE structure. BLOB fields are numbered positively. BLOB declarations begin with 1 and increment by 1 for each subsequent BLOB, in the order in which they appear within the FILE structure.

Multi-Use File Structure Properties

PROP:Label

Returns the label of a declaration statement.

When no array element number is specified and the *target* is the label of a KEY (or INDEX), PROP:Label returns the label of the KEY (or INDEX).

When a positive array element number is specified and the *target* is a FILE, PROP:Label returns the label of the specified field within the RECORD structure.

When a negative array element number is specified and the *target* is a FILE, PROP:Label returns the label of the specified MEMO within the FILE structure.

When a positive array element number is specified and the *target* is a BLOB, PROP:Label returns the label of the specified BLOB.

PROP:NAME

The NAME attribute of the declaration statement.

When no array element number is specified and the *target* is the label of a FILE, PROP:Name returns the contents of the FILE statement's NAME attribute.

When a positive array element number is specified and the *target* is the label of a FILE, PROP:Name returns the NAME attribute of the specified field within the RECORD structure.

When a negative array element number is specified and the *target* is the label of a FILE, PROP:Name returns the NAME attribute of the specified MEMO within the FILE structure.

When no array element number is specified and the *target* is the label of a KEY (or INDEX), PROP:Name returns the NAME attribute of the specified KEY (or INDEX).

When a positive array element number is specified and the *target* is a BLOB, PROP:Name returns the NAME attribute of the specified BLOB.

PROP:Type

The data type of the declaration statement.

When no array element number is specified and the *target* is the label of a KEY (or INDEX), PROP:Type returns either "KEY" or "INDEX."

When a positive array element number is specified and the *target* is the label of a FILE, PROP:Type returns the data type of the specified field within the RECORD structure.

FILE Statement Properties

These properties all take the label of a FILE as their *target*.

PROP:DRIVER

The DRIVER attribute. Returns the file driver of the FILE.

PROP:DriverString

A FILE property that returns the second parameter of the DRIVER() attribute of a file.

PROP:CREATE

The CREATE attribute on the FILE statement. A toggle attribute which contains a null string (") if absent, and '1' if present.

PROP:RECLAIM

The RECLAIM attribute on the FILE statement. A toggle attribute which contains a null string (") if absent, and '1' if present.

PROP:OWNER

The OWNER attribute on the FILE statement.

PROP:ENCRYPT

The ENCRYPT attribute on the FILE statement. A toggle attribute which contains a null string (") if absent, and '1' if present.

PROP:THREAD

The THREAD attribute on the FILE statement. A toggle attribute which contains a null string (") if absent, and '1' if present.

PROP:OEM

The OEM attribute on the FILE statement. A toggle attribute which contains a null string (") if absent, and '1' if present.

PROP:Keys

Returns the number of KEY and INDEX declarations in the FILE structure.

PROP:Key

An array that returns a reference to the specified KEY or INDEX in the FILE structure. This reference can be used as the source side of a reference assignment statement.

Key Properties

These properties all take the label of a KEY (or INDEX) as their *target*.

PROP:PRIMARY

The PRIMARY attribute on the KEY statement. A toggle attribute which contains a null string (") if absent, and '1' if present.

PROP:DUP

The DUP attribute on the KEY statement. A toggle attribute which contains a null string (") if absent, and '1' if present.

PROP:NOCASE

The NOCASE attribute on the KEY or INDEX statement. A toggle attribute which contains a null string (") if absent, and '1' if present.

PROP:OPT

The OPT attribute on the KEY or INDEX statement. A toggle attribute which contains a null string (") if absent, and '1' if present.

PROP:Components

Returns the number of component fields of a KEY or INDEX.

PROP:Field

An array that returns the field number (within the RECORD structure) of the specified component field of a KEY or INDEX. This field number can be used as the array element number for PROP:Label or PROP:Name.

PROP:Ascending

An array that returns '1' if the specified key component is in ascending order, and a null string (") if in descending order.

Field Properties

These properties all take the label of a FILE as their *target*.

PROP:Memos

Returns the number of MEMO fields in the FILE structure.

PROP:Blobs

Returns the number of BLOB fields in the FILE structure.

PROP:BINARY

The BINARY attribute on the MEMO or BLOB statement in the FILE structure. A toggle attribute which contains a null string (") if absent, and '1' if present.

PROP:Fields

Returns the number of fields declared in the RECORD structure.

PROP:Size

An array that returns the declared size of the specified MEMO, STRING, CSTRING, PSTRING, DECIMAL, or PDECIMAL field.

PROP:Places

An array that returns the number of decimal places declared for the specified DECIMAL or PDECIMAL field.

PROP:Dim

An array property of a file that returns the product of the array dimensions specified in the DIM attribute of the specified field. For example, for a field DIM(3,2) PROP:Dim returns 6.

PROP:Over

An array property of a file that returns the field number of the field referenced in the OVER attribute on the specified field.

Example:

```

PROGRAM
MAP
PrintFile          PROCEDURE(*FILE F)
DumpGroupDetails  PROCEDURE(USHORT start, USHORT total)
DumpFieldDetails  PROCEDURE(USHORT indent, USHORT FieldNo)
DumpToFile        PROCEDURE
SetAttribute      PROCEDURE(SIGNED Prop,STRING Value)
StartLine         PROCEDURE(USHORT indent,STRING label, STRING type)
Concat            PROCEDURE(STRING s)
END

LineSize          EQUATE(255)
FileIndent        EQUATE(20)

DestName          STRING(FILE:MaxFilePath)

DestFile          FILE,DRIVER('ASCII'),CREATE,NAME(DestName)
Record            RECORD
Line              STRING(LineSize)
END
END

Employee          FILE,DRIVER('TOPSPEED'),NAME('Employee.tps'),PRE(EMP),BINDABLE,CREATE,THREAD
EmpID_Key         KEY(EMP:EmpID),PRIMARY
EmpName_Key       KEY(EMP:Lname,EMP:Fname,EMP:MInit),DUP
JobID_Key         KEY(EMP:JobID),DUP
PubID_Key         KEY(EMP:PubID),DUP
DateKey          KEY(-EMP:Hire_date),DUP,NOCASE,OPT
Record           RECORD,PRE()
EmpID            CSTRING(10)
Fname            CSTRING(21)
MInit           CSTRING(2)
Lname            CSTRING(31)
JobID            SHORT
Job_lvl          BYTE
PubID            CSTRING(5)
Hire_date        DATE
PictureFile      STRING(65)
END
END

TheFile &FILE
AKey            &KEY
Line            STRING(LineSize)
Blobs           LONG

```

```

CODE
PrintFile(Employee)

PrintFile PROCEDURE(*FILE F)
CODE
IF NOT FILEDIALOG('Choose Output File',DestName,'Text|*.TXT|Source|*.CLW',0100b)
RETURN
END
OPEN(DestFile)
IF ERRORCODE()
CREATE(DestFile)
OPEN(DestFile)
END
ASSERT(ERRORCODE()=0)

TheFile &= F
DO DumpFileDetails
DO DumpKeys
DO DumpMemos
DumpGroupDetails(0, F{PROP:Fields})
StartLine(FileIndent,',' , 'END')
DumpToFile

DumpFileDetails ROUTINE
StartLine(FileIndent,TheFile{PROP:label},'FILE')
Concat(',DRIVER('' & CLIP(TheFile{PROP:Driver}))')
IF TheFile{PROP:DriverString}
Concat(', ' & CLIP(TheFile{PROP:DriverString}))')
END
Concat('')')
SetAttribute(TheFile{PROP:Create},'CREATE')
SetAttribute(TheFile{PROP:Reclaim},'RECLAIM')
IF TheFile{PROP:Owner}
Concat(',OWNER('' & CLIP(TheFile{PROP:Owner}) & ''')')
END
SetAttribute(TheFile{PROP:Encrypt},'ENCRYPT')
Concat(',NAME('' & CLIP(TheFile{PROP:Name}) & ''')')
SetAttribute(TheFile{PROP:Thread},'THREAD')
SetAttribute(TheFile{PROP:OEM},'OEM')
DumpToFile

```

```

DumpMemos ROUTINE
LOOP X# = 1 TO TheFile{PROP:Memos}
  StartLine(FileIndent+2,TheFile{PROP:label,-X#},'MEMO(')
  Concat(CLIP(TheFile{PROP:Size,-X#})&')')
  SetAttribute(TheFile{PROP:Binary,-X#},'BINARY')
  IF TheFile{PROP:Name,-X#}
    Concat(',NAME(' & CLIP(TheFile{PROP:Name,-X#}) & ')')
  END
  DumpToFile
END

DumpKeys ROUTINE
LOOP X# = 1 TO TheFile{PROP:Keys}
  AKey &= TheFile{PROP:Key,X#}
  StartLine(FileIndent+2,AKey{PROP:label},AKey{PROP:Type})
  Concat('(')
  LOOP Y# = 1 TO AKey{PROP:Components}
    IF Y# > 1 THEN Concat(',').
    IF AKey{PROP:Ascending,Y#}
      Concat('+')
    ELSE
      Concat('-')
    END
    Concat(TheFile{PROP:Label,akey{PROP:Field,Y#}})
  END
  Concat(')')
  SetAttribute(AKey{PROP:Dup},'DUP')
  SetAttribute(AKey{PROP:NoCase},'NOCASE')
  SetAttribute(AKey{PROP:Opt},'OPT')
  SetAttribute(AKey{PROP:Primary},'PRIMARY')
  IF AKey{PROP:Name}
    Concat(',NAME(' & CLIP(AKey{PROP:Name}) & ')')
  END
  DumpToFile
END

DumpGroupDetails PROCEDURE(USHORT start, USHORT total)
fld          USHORT
fieldsInGroup USHORT
GroupIndent  USHORT,STATIC
CODE
IF start = 0 THEN
  GroupIndent = FileIndent+2
  StartLine(GroupIndent,'RECORD','RECORD')
  DumpToFile
END
GroupIndent += 2
LOOP fld = start+1 TO start+total

```

```

DumpFieldDetails(GroupIndent, fld)
IF TheFile{PROP:Type, fld} = 'GROUP'
  fieldsInGroup = TheFile{PROP:Fields, fld}
  DumpGroupDetails (fld, fieldsInGroup)
  fld += fieldsInGroup
END
END
GroupIndent -= 2
StartLine(GroupIndent, '', 'END')
DumpToFile

DumpFieldDetails PROCEDURE(USHORT indent, USHORT FieldNo)
FldType STRING(20)
CODE
FldType = TheFile{PROP:Type, FieldNo}
StartLine(indent, TheFile{PROP:Label, FieldNo}, FldType)
IF INSTR('STRING', FldType, 1, 1) OR INSTR('DECIMAL', FldType, 1, 1)
  Concat('(' & TheFile{PROP:Size, FieldNo})
  IF FldType = 'DECIMAL' OR FldType = 'PDECIMAL'
    Concat(', ' & TheFile{PROP:Places, FieldNo})
  END
  Concat(')')
END
IF TheFile{PROP:Dim, FieldNo} <> 0
  Concat(', DIM(' & CLIP(TheFile{PROP:Dim, FieldNo}) & ')')
END
IF TheFile{PROP:Over, FieldNo} <> 0
  Concat(', OVER(' & CLIP(TheFile{PROP:Label, TheFile{PROP:Over, FieldNo}}) & ')')
END
IF TheFile{PROP:Name, FieldNo}
  Concat(', NAME(' & CLIP(TheFile{PROP:Name, FieldNo}) & ')')
END
DumpToFile

SetAttribute PROCEDURE (Prop, Value)
CODE
IF Prop THEN Line = CLIP(Line) & ', ' & CLIP(Value).

```

```
StartLine PROCEDURE (USHORT indent,STRING label, STRING type)
```

```
TypeStart USHORT
```

```
CODE
```

```
Line = label
```

```
IF LEN(CLIP(Line)) < Indent
```

```
    TypeStart = Indent
```

```
ELSE
```

```
    TypeStart = LEN(CLIP(Line)) + 4
```

```
END
```

```
Line[TypeStart : LineSize] = type
```

```
Concat PROCEDURE (STRING s)
```

```
CODE
```

```
Line = CLIP(Line) & s
```

```
DumpToFile PROCEDURE
```

```
CODE
```

```
DestFile.Line = Line; ADD(DestFile)
```

```
ASSERT(ERRORCODE()=0)
```

Environment Files

An environment file contains internationalization settings for an application. On program initialization, the Clarion run-time library attempts to locate an environment file with the same name and location as your application's program file (*appname*.ENV). If an environment file is not found, the run-time library defaults to standard English/ASCII. You can also use these settings to specify internationalization issues for the Clarion environment by creating a CLARION6.ENV file (the Database Manager uses these settings when displaying data files).

The .ENV file is compatible with the .INI files used by Clarion for DOS (both versions 3 and 3.1) if the CLACHARSET is set to OEM, because Clarion for DOS .INI files are generally written using OEM ASCII, not the ANSI character set.

The LOCALE procedure can be used to load environment files at run-time to dynamically change the international settings. LOCALE can also be used to set individual entries. International support is dependent on support in the File Driver (generally for the OEM attribute); consult the File Driver documentation for information on international support in specific drivers.

The following settings can be set in an environment file:

CLASYSTEMCHARSET=WINDOWS

CLASYSTEMCHARSET="charset"

Specifies the value of the system charset. If parameter is not set or it is set to WINDOWS, the system charset is equivalent to CHARSET:ANSI. Setting this in the .ENV file is equivalent to setting the property SYSTEM{PROP:CharSet}.

CLACHARSET=WINDOWS

CLACHARSET=OEM

This determines the character set used by the entries in the .ENV file. WINDOWS is the default if this setting is omitted from the environment file. Use the OEM setting if you are using a DOS editor to edit the .ENV file, or if it has to be compatible with Clarion for DOS. Otherwise, specify WINDOWS or omit the entry. This should always be the first setting in the environment file.

CLACOLSEQ=WINDOWS

CLACOLSEQ="string"

Specifies a specific collating sequence for use at run-time. This collating sequence is used for building KEY and INDEX files, as well as for sorting QUEUES and all string/character comparisons.

If the WINDOWS setting is used, then the default collation sequence is defined by Windows' Country setting (in the Control Panel). If this entry is omitted from the environment file, then the default ANSI ordering is used, not the windows default.

Using the WINDOWS setting, the ordering can 'interleave' characters of differing case (AaBbCc ...), so code such as

```
CASE SomeString[1]
OF 'A' TO 'Z'
```

includes 'a' TO 'y' as well. Use the ISUPPER and ISLOWER procedures in preference to this kind of code if WINDOWS (or other non-default) collation sequences are used.

In addition to the WINDOWS setting, you may specify a *string* of characters (in double quotes) to explicitly define the collation sequence to use. Only those characters that need to have their sort order specified need be included; all other characters not listed remain in their same relative order. For example, if CLACOLSEQ="CA" is specified for the standard English sort (ABCD ...) the resulting sort order is "CBAD." This is a change from the Clarion for DOS versions of this setting that needed exactly 222 characters, but it is backward compatible.

NOTE: You should always read and write files using the same collation sequence. Using a different sequence may result in keys becoming out of order and records becoming inaccessible. Specifying CLACOLSEQ=WINDOWS means that the collation sequence may change if the user changes the Country in Windows' Control Panel. If the collation sequence changes, use BUILD to rebuild the keys in your data files.

CLAAMPM=WINDOWS

CLAAMPM="AMstring", "PMstring"

This specifies the text used to indicate AM or PM as a part of a time display field. The WINDOWS setting specifies use of the AM/PM strings set up in the Windows Control Panel. The *AMstring* and *PMstring* settings are the same as in Clarion for DOS, except that they take notice of the setting of CLACHARSET.

CLAMONTH="Month1", "Month2", ... , "Month12"

Specifies the text returned by procedures and picture formats involving the month full name.

CLAMON="AbbrevMonth1", "AbbrevMonth2", ... , "AbbrevMonth12"

Specifies the text returned by procedures and picture formats involving the abbreviated month name.

CLADIGRAPH="DigraphChar1Char2, ... "

This allows *Digraph* characters to collate correctly. A *Digraph* is a single logical character that is a combination of two characters (*Char1* and *Char2*). The *Digraph* is collated as the two characters that combine to create it. They are more common in non-English languages. For example, with CLADIGRAPH="ÆAe,æae" specified, the word "Jæger" sorts before "Jager" (since "Jae" comes before "Jag"). Multiple *DigraphChar1Char2* combinations may be defined, separated by commas. This setting takes notice of the CLACHARSET setting.

CLACASE=WINDOWS**CLACASE="UpperString", "LowerString"**

Allows you to specify upper and lower case letter pairs.

The WINDOWS setting uses the default upper/lower case pair sets as defined by the Windows Country setting (in the Control Panel). If this entry is omitted from the environment file, then the default ANSI ordering is used, not the windows default.

The *UpperString* and *LowerString* parameters specify a set of uppercase characters and each one's lowercase equivalent. The length of the *UpperString* and *LowerString* parameters must be equal. CLACASE takes notice of the setting of CLACHARSET. ANSI characters less than 127 are not affected.

CLABUTTON="OK", "&Yes", "&No", "&Abort", "&Retry", "&Ignore", Cancel", "&Help"

This defines the text used by the buttons of the MESSAGE procedure. The text is specified as a list of comma separated strings in the following order: OK, YES, NO, ABORT, RETRY, IGNORE, CANCEL, HELP. The default is as specified above.

CLAMSGerrornumber="ErrorMessage"

This allows run-time error messages to be overridden with translated strings. The *errornumber* is a standard Clarion error code number appended to CLAMSG. *ErrorMessage* is the string value used to replace that error number's default message. For example, CLAMSG2="No File Found" makes "No File Found" the return value of the ERROR() procedure when ERRORCODE() = 2.

CLALFN=OFF

This disables use of long filenames in the program.

Example:

```
CLACHARSET=WINDOWS
CLACOLSEQ="AÄÅÆaàáâãäåæBbCcçDdEēĒeëĕĚFfGgHhIiĭĳjKkLlMmNñňOoōöôPpQqRrSsTtUūuüüVvWwXxYyZzÿ"
CLAAMPM="AM", "PM"
CLAMONTH="January", "February", "March", "April", "May", "June", "July", "August", "September", "October", "November", "December"
CLAMON="Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
CLADIGRAPH="ÆAe, æae"
CLACASE="ÄÅÆĈĖŃÖ", "äåæçéñö"
CLABUTTON="OK", "&Si", "&No", "&Abortar", "&Volveratratar", "&Ignora", "Cancelar", "&Ayuda"
CLAMSG2="No File Found"
```

View Structures

VIEW (declare a "virtual" file)

```

label  VIEW(primary file) [,FILTER( )] [,ORDER( )]
        [PROJECT( )]
        [JOIN( )
          [PROJECT( )]
          [JOIN( )
            [PROJECT( )]
          ]
        ]
        END]
      END
  
```

VIEW	Declares a "virtual" file as a composite of related files.
<i>label</i>	The name of the VIEW.
<i>primary file</i>	The label of the primary FILE of the VIEW.
FILTER	Declares an expression used to filter valid records for the VIEW (PROP:FILTER).
ORDER	Declares an expression or list of expressions used to define the sorted order of records for the VIEW (PROP:ORDER or PROP:SQLOrder).
PROJECT	Specifies the fields from the <i>primary file</i> , or the secondary related file specified by a JOIN structure, that the VIEW will retrieve. If omitted, all fields from the file are retrieved.
JOIN	Declares a secondary related file.

VIEW declares a "virtual" file as a composite of related data files. The data elements declared in a VIEW do not physically exist in the VIEW, because the VIEW structure is a logical construct. VIEW is a separate method of addressing data physically residing in multiple, related FILE structures. At run-time, the VIEW structure is not assigned memory for a data buffer, so the fields used in the VIEW are placed in their respective FILE structure's record buffer.

A VIEW structure must be explicitly OPENed before use, and all primary and secondary related files used in the VIEW must have been previously OPENed.

Either a SET statement on the VIEW's primary file before the OPEN(view), or a SET(view) statement after the OPEN(view), must be issued to set the VIEW's processing order and starting point, then NEXT(view) or PREVIOUS(view) allow sequential access to the VIEW.

The VIEW data structure is designed for sequential access, but also allows random access using the REGET statement. The REGET statement is also available for VIEW, but only to specify the primary and secondary related file records that should be current in their respective record buffers after the VIEW is CLOSED. If no REGET statement is issued immediately before the CLOSE(view) statement, the primary and secondary related file record buffers are set to no current record.

The processing sequence of the primary and secondary related files is undefined after the VIEW is CLOSED. Therefore, SET or RESET must be used to establish sequential file processing order, if necessary, after closing the VIEW.

The VIEW data structure is designed to facilitate database access on client-server systems. It accomplishes two relational operations at once: the relational "Join" and "Project" operations. On client-server systems, these operations are performed on the file server, and only the result of the operation is sent to the client. This can dramatically improve performance of network applications.

A relational "Join" retrieves data from multiple files, based upon the relationships defined between the files. The JOIN structure in a VIEW structure defines the relational "Join" operation. There may be multiple JOIN structures within a VIEW, and they may be nested within each other to perform multiple-level "Join" operations. The VIEW structure defaults to a "left outer join," where all records for the VIEW's *primary file* are retrieved whether the secondary file named in a JOIN structure contains any related records or not. The secondary file fields are implicitly CLEARed (zero or blank) for those primary file records without related secondary records. You can override the default left outer join by specifying the INNER attribute on the JOIN (creating an "inner join") so that only those *primary file* records with related secondary file records are retrieved.

A relational "Project" operation retrieves only specified data elements from the files involved, not their entire record structure. Only those fields explicitly declared in PROJECT statements in the VIEW structure are retrieved if there are any PROJECT statements declared. Therefore, the relational "Project" operation is automatically implemented by the VIEW structure. The contents of any fields that are not contained in PROJECT statements are undefined.

The FILTER attribute restricts the VIEW to a sub-set of records. The FILTER expression may include any of the fields explicitly declared in the VIEW structure and restrict the VIEW based upon the contents of any of the fields. This makes the FILTER operate across all levels of the "Join" operation.

NOTE:

VIEWS have no THREAD attribute by syntax, but VIEWS declared in the local scope of a PROCEDURE or ROUTINE are treated as threaded. A VIEW declared in the global or module scope is treated as threaded if at least one joined FILE is threaded.

Related Procedures: BUFFER, CLOSE, FLUSH, OPEN, RECORDS, DELETE, HOLD, NEXT, POSITION, PREVIOUS, PUT, RELEASE, REGET, RESET, SET, SKIP, WATCH

Example:

```
Customer  FILE,DRIVER('Clarion'),PRE(Cus) !Declare customer file layout
AcctKey   KEY(Cus:AcctNumber)
Record    RECORD
AcctNumber LONG
OrderNumber LONG
Name      STRING(20)
Addr      STRING(20)
City      STRING(20)
State     STRING(20)
Zip       STRING(20)
          END
```

```
Header    FILE,DRIVER('Clarion'),PRE(Hea) !Declare header file layout
AcctKey   KEY(Hea:AcctNumber)
OrderKey  KEY(Hea:OrderNumber)
Record    RECORD
AcctNumber LONG
OrderNumber LONG
ShipToName STRING(20)
ShipToAddr STRING(20)
ShipToCity STRING(20)
ShipToState STRING(20)
ShipToZip  STRING(20)
          END
          END
```

```
Detail    FILE,DRIVER('Clarion'),PRE(Dt1) !Declare detail file layout
OrderKey  KEY(Dt1:OrderNumber)
Record    RECORD
OrderNumber LONG
Item      LONG
Quantity  SHORT
          END
          END
```

```
Product   FILE,DRIVER('Clarion'),PRE(Pro) !Declare product file layout
ItemKey   KEY(Pro:Item)
Record    RECORD
Item      LONG
Description STRING(20)
Price     DECIMAL(9,2)
          END
          END
```

```
ViewOrder VIEW(Customer)           !Declare VIEW structure
  PROJECT(Cus:AcctNumber,Cus:Name)
  JOIN(Hea:AcctKey,Cus:AcctNumber)  !Join Header file
  PROJECT(Hea:OrderNumber)
  JOIN(Dtl:OrderKey,Hea:OrderNumber) !Join Detail file
  PROJECT(Det:Item,Det:Quantity)
  JOIN(Pro:ItemKey,Dtl:Item)        !Join Product file
  PROJECT(Pro:Description,Pro:Price)
  END
  END
  END
  END
```

See Also:

JOIN

PROJECT

PROJECT (set view fields)**PROJECT(*fields*)**

PROJECT Declares the fields retrieved for the VIEW.

fields A comma delimited list of fields (including prefixes) from the primary file of the VIEW, or the secondary related file named in the JOIN structure, containing the PROJECT declaration.

The **PROJECT** statement declares *fields* retrieved for a relational "Project" operation. A relational "Project" operation retrieves only the specified *fields* from the file, not the entire record structure.

A PROJECT statement may be declared in the VIEW, or within one of its component JOIN structures. If there is no PROJECT declaration in the VIEW or JOIN structure, all fields in the relevant file are retrieved.

If a PROJECT statement is present in the VIEW or JOIN structure only the fields explicitly declared in the PROJECT are guaranteed to be retrieved. The contents of any fields that are not contained in PROJECT statements are undefined. Depending on the abilities of the particular database engine you are using, other fields may be retrieved. However, you should not rely on this as future changes or changes in the database driver may preclude these fields from being retrieved.

Example:

```

Detail  FILE,DRIVER('Clarion'),PRE(Dt1) !Declare detail file layout
OrderKey  KEY(Dt1:OrderNumber)
Record    RECORD
OrderNumber  LONG
Item        LONG
Quantity    SHORT
Description  STRING(20) !Line item comment
          END
          END
Product  FILE,DRIVER('Clarion'),PRE(Pro) !Declare product file layout
ItemKey   KEY(Pro:Item)
Record    RECORD
Item      LONG
Description  STRING(20) !Product description
Price     DECIMAL(9,2)
          END
          END
ViewOrder VIEW(Detail)
          PROJECT(Det:OrderNumber,Det:Item,Det:Description)
          JOIN(Pro:ItemKey,Det:Item)
          PROJECT(Pro:Description,Pro:Price)
          END
          END

```

JOIN (declare a "join" operation)

```

JOIN( | secondary key ,linking fields | ) [, INNER ]
      | secondary file ,expression |
      [PROJECT( )]
      [JOIN( )
        [PROJECT( )]
      ]
      END]
END

```

JOIN	Declares a secondary file for a relational "Join" operation.
<i>secondary key</i>	The label of a KEY which defines the secondary FILE and its access key.
<i>linking fields</i>	A comma-delimited list of fields in the related file that contain the values the <i>secondary key</i> uses to get records.
<i>secondary file</i>	The label of the secondary FILE.
<i>expression</i>	A string constant containing a single logical expression for joining the files (PROP:JoinExpression or PROP:SQLJoinExpression). This expression may include any of the logical and Boolean operators.
INNER	Specifies an "inner join" instead of the default "left outer join"--the only records retrieved from the VIEW's <i>primary file</i> parent are those with at least one related record in the JOIN's <i>secondary file</i> .
PROJECT	Specifies the fields from the secondary related file specified by a JOIN structure that the VIEW will retrieve. If omitted, all fields from the file are retrieved.

The **JOIN** structure declares a secondary file for a relational "Join" operation. A relational "Join" retrieves data from multiple files, based upon the relationships defined between the files. There may be multiple JOIN structures within a VIEW, and they may be nested within each other to perform multiple-level "Join" operations.

The *secondary key* defines the access key for the secondary file. The *linking fields* name the fields in the file to which the secondary file is related, that contain the values used to retrieve the related records. For a JOIN directly within the VIEW, these fields come from the VIEW's primary file. For a JOIN nested within another JOIN, these fields come from the secondary file of the JOIN in which it is nested. Non-linking fields in the *secondary key* are allowed as long as they appear in the list of the key's component fields after all the linking fields.

When data is retrieved, if there are no matching secondary file records for a primary file record, blank or zero values are supplied in the fields specified in the PROJECT. This type of relational "Join" operation is known as a "left outer join."

The *expression* parameter allows you to join files which contain related fields but no keys defined for the relationship. PROP:JoinExpression and PROP:SQLJoinExpression are array properties whose the array element number references the ordinal position of the JOIN in the VIEW to affect. PROP:SQLJoinExpression is an SQL-only version of PROP:JoinExpression. If the first character of the expression assigned to PROP:JoinExpression or PROP:SQLJoinExpression is a plus sign (+) the new expression is concatenated to the existing join expression.

Example:

```
Customer  FILE,DRIVER('Clarion'),PRE(Cus) !Declare customer file layout
AcctKey   KEY(Cus:AcctNumber)
Record    RECORD
AcctNumber LONG
OrderNumber LONG
Name      STRING(20)
          END
          END
```

```
Header    FILE,DRIVER('Clarion'),PRE(Hea) !Declare header file layout
AcctKey   KEY(Hea:AcctNumber)
OrderKey  KEY(Hea:AcctNumber,Hea:OrderNumber)
Record    RECORD
AcctNumber LONG
OrderNumber LONG
Total     DECIMAL(11,2)    !Total cash paid
Discount  DECIMAL(11,2)    !Discount amount given
OrderDate LONG
          END
          END
```

```
Detail    FILE,DRIVER('Clarion'),PRE(Dt1) !Declare detail file layout
OrderKey  KEY(Dt1:AcctNumber,Dt1:OrderNumber)
Record    RECORD
AcctNumber LONG
OrderNumber LONG
Item      LONG
Quantity  SHORT
          END
          END
```

```
Product    FILE,DRIVER('Clarion'),PRE(Pro) !Declare product file layout
ItemKey    KEY(Pro:Item)
Record     RECORD
Item       LONG
Description STRING(20)
Price      DECIMAL(9,2)
           END
           END

ViewOrder1 VIEW(Header)                                !Declare VIEW structure
           PROJECT(Hea:AcctNumber,Hea:OrderNumber)
           JOIN(Dtl:OrderKey,Hea:AcctNumber,Hea:OrderNumber) !Join Detail file
           PROJECT(Dtl:ItemDtl:Quantity)
           JOIN(Pro:ItemKey,Dtl:Item)                    !Join Product file
           PROJECT(Pro:Description,Pro:Price)
           END
           END

ViewOrder2 VIEW(Customer)                              !Declare VIEW structure
           JOIN(Header,'Cus:AcctNumber = Hea:AcctNumber AND ' & |
               ' (Hea:Discount + Hea:Total) * .1 > Hea:Discount')
           PROJECT(Hea:AcctNumber,Hea:OrderNumber)
           JOIN(Dtl:OrderKey,Hea:AcctNumber,Hea:OrderNumber) !Join Detail file
           PROJECT(Dtl:ItemDtl:Quantity)
           END
           END
```

See Also:

INNER

Queue Structures

QUEUE (declare a memory QUEUE structure)

```
label  QUEUE( [group] ) [,PRE] [,STATIC] [,THREAD] [,TYPE] [,BINDABLE] [,EXTERNAL] [,DLL]
```

```
fieldlabel variable [,NAME( )]
```

```
      END
```

QUEUE	Declares a memory queue structure.
<i>label</i>	The name of the QUEUE.
<i>group</i>	The label of a previously declared GROUP or QUEUE structure from which it will inherit its structure. This may be a GROUP or QUEUE with or without the TYPE attribute.
PRE	Declare a <i>fieldlabel</i> prefix for the structure.
STATIC	Declares a QUEUE, local to a PROCEDURE, whose buffer is allocated in static memory.
THREAD	Specify memory for the queue is allocated once for each execution thread. This implies the STATIC attribute on Procedure Local data.
TYPE	Specify the QUEUE is just a type definition for other QUEUE declarations.
BINDABLE	Specify all variables in the queue may be used in dynamic expressions.
EXTERNAL	Specify the QUEUE is defined, and its memory is allocated, in an external library.
DLL	Specify the QUEUE is defined in a .DLL. This is required in addition to the EXTERNAL attribute.
<i>fieldlabel</i>	The name of the <i>variables</i> in the queue.
<i>variable</i>	Data declaration. The sum of the memory required for all declared <i>variables</i> in the QUEUE must not be greater than 4MB.

QUEUE declares a memory QUEUE structure. The *label* of the QUEUE structure is used in queue processing statements and procedures. When used in assignment statements, expressions, or parameter lists, a QUEUE is treated like a GROUP data type.

The structure of a QUEUE declared with the *group* parameter begins with the same structure as the named *group*; the QUEUE inherits the fields of the named *group*. The QUEUE may also contain its own *declarations* that follow the inherited fields. If the QUEUE will not contain any other fields, the name of the *group* from which it inherits may be used as the data type without the QUEUE or END keywords.

A QUEUE may be thought of as a "memory file" internally implemented as a "dynamic array" of QUEUE entries. When a QUEUE is declared, a data buffer is allocated (just as with a file). Each entry in the QUEUE is run-length compressed during an ADD or PUT to occupy as little memory as necessary, and de-compressed during GET. There is an 8 byte per-entry overhead for queues with uncompressed records, and 12 bytes per entry for queues with compressed records.

The data buffer for a Procedure local QUEUE (declared in the data section of a PROCEDURE) is allocated on the stack (unless it has the STATIC attribute or is too large). The memory allocated to the entries in a procedure-local QUEUE without the STATIC attribute is allocated only until you FREE the QUEUE, or you RETURN from the PROCEDURE--the QUEUE is automatically FREED upon RETURN.

For a Global data, Module data, or Local data QUEUE with the STATIC attribute, the data buffer is allocated static memory and the data in the buffer is persistent between procedure calls. The memory allocated to the entries in the QUEUE remains allocated until you FREE the QUEUE.

The *variables* in the QUEUE's data buffer are not automatically initialized to any value, they must be explicitly assigned values. Do not assume that they contain blanks or zero before your program's first assignment to them.

As entries are added to the QUEUE, memory for the entry is dynamically allocated then the data copied from the buffer to the entry and compressed. As entries are deleted from the QUEUE, the memory used by the deleted entry is freed. The maximum number of entries in a QUEUE is theoretically 2^{26} (67,108,864), but is actually dependant upon available virtual memory. The actual memory used by each entry in the QUEUE is dependent on the data compression ratio achieved by the runtime library.

A QUEUE with the BINDABLE attribute makes all the variables within the QUEUE available for use in a dynamic expression, without requiring a separate BIND statement for each (allowing BIND(queue) to enable all the fields in the queue). The contents of each variable's NAME attribute is the logical name used in the dynamic expression. If no NAME attribute is present, the label of the variable (including prefix) is used. Space is allocated in the .EXE for the names of all of the variables in the structure. This creates a larger program that uses more memory than it normally would. Therefore, the BINDABLE attribute should only be used when a large proportion of the constituent fields are going to be used.

A QUEUE with the TYPE attribute is not allocated any memory; it is only a type definition for QUEUES that are passed as parameters to PROCEDURES. This allows the receiving procedure to directly address component fields in the passed QUEUE. The parameter declaration on the PROCEDURE statement instantiates a local prefix for the passed QUEUE as it names the passed QUEUE for the procedure. For example, PROCEDURE(LOC:PassedGroup) declares the procedure uses the LOC: prefix (along with the individual field names used in the type declaration) to directly address component fields of the QUEUE actually passed as the parameter.

The WHAT and WHERE procedures allow access to the fields by their relative position within the QUEUE structure.

Related Procedures:

ADD, CHANGES, DELETE, FREE, GET, POINTER,
POSITION
PUT, RECORDS, SORT

See Also:

PRE
STATIC
NAME
FREE
THREAD
WHAT
WHERE

Example:

```
NameQue  QUEUE,PRE(Nam)           !Declare a queue
Name      STRING(20)
Zip       DECIMAL(5,0),NAME('SortField')
          END                     !End queue structure

NameQue2  QUEUE(NameQue),PRE(Nam2) !Queue that inherits Name and Zip fields
Phone     STRING(10)              ! and adds a Phone field
          END

NameQue3  NameQue2                !Declare a second QUEUE with exactly
          ! the same structure as NameQue2
```

Additional QUEUE Considerations

This topic expands on important issues in understanding how supported processing statements and optional parameters can affect a QUEUE.

ADD, GET, PUT to a QUEUE by KEY

There are three (3) forms of the QUEUE *key* parameter: *sequence*, *name* and *function*. Each one is described as follows:

sequence [\pm]key1[, [\pm]key2...]

A list of up to 16 labels of QUEUE fields, separated by commas and with an optional + or - sign preceding the label. If the key component is specified with -, it is used in descending order. Reference field types (including ANY) and arrays are not permitted.

name

A string constant, variable or expression. Its value must contain a list of up to 16 NAME attributes of QUEUE fields with an optional leading + or - sign, separated with commas. If the key component is specified with -, it is used in descending order. Reference field types (including ANY) and arrays are not permitted.

When comparing two keys using one of the two forms above, the keys are considered *equal* if all key components are equal.

A key is considered *greater than* another key, if its *n*th key component is greater than the second key, and the extra component defined is in ascending order (with all other key components being equal.)

Finally, a key is considered *less than* another key if its *n*th key component is greater than the second key, and the extra component defined is in descending order (with all other key components being equal.)

There is also a third form of KEY types regarding QUEUES:

function

The label of the function containing two parameters of a *GROUP or named GROUP passed by address, and having a SIGNED return value. Both parameters must use the same parameter type, and cannot be omitted. The RAW, C and PASCAL attributes are not permitted in the prototype declaration.

The first parameter of the FUNCTION is the *target* parameter, or record that is about to be acted on. The second parameter is a *comparison* value, used to determine the position where the first parameter is to be placed or retrieved.

Using ADD, PUT or GET by FUNCTION will read or write from a positional value returned by the function.

If the function returns zero (0) the queue record of the first parameter is treated as equal to the second. In this case, no record is ADDED or PUT, since the values are equal.

If the function returns a negative value, the ADD or PUT of the record passed as a first parameter is treated as having less value than record passed as second parameter and is written accordingly.

If the function returns a positive value, the ADD or PUT of the record passed as a first parameter is treated as having a greater value than record passed as second parameter and is written accordingly.

Using Multiple Sort Orders

The following topic describes the internal paradigm regarding QUEUES with multiple sort orders.

Prior to Clarion 5, it was possible to use the GET(Queue,Key) form to obtain a POINTER to the first or last record in a range.

For example:

```

Q  QUEUE
A   LONG
B   STRING(20)
END

CODE
...
SORT(Q, Q.A, Q.B)      !sort the queue in a,b sequence
Q.A = 1                !set to the first record
CLEAR (Q.B)           !clear secondary field to make sure
GET (Q, Q.A)          !GET to first record
first# = POINTER(Q)   !GET fails, but stores the POINTER
                       !where record would have existed

Q.A = 5                !set to the last record or beyond
CLEAR (Q.B)           !again, clear secondary sort field
GET (Q, Q.A)          !GET will fail
last# = POINTER(Q)-1  !and returns where new record will be

```

After this code executes, *first#* contains a pointer to the first record of the **Q.A** field in a range (in this example range is 1 through 4), and *last#* contains a pointer to the last record with value of **Q.A** within this range.

This technique is not possible for queues with multiple sort orders. Even if one set of key components is a subset of another one, sort orders based on them are handled separately.

Hence, if the program uses a partial key seed value in the GET(Queue,Key) statement, the queue logic must build the sort order based on specified key if it does not exist, and perform the GET operation using this sort order. In other words, the GET never fails.

Sort orders based on "full" and "partial" keys can be different because of queue rules: ADD adds a new record after all other records with the same key, and PUT updates an existing record after all other records with the same key. The new POSITION(Queue) function implements the behavior that an attempted GET on a partial key value used to have.

In Clarion 6, every active QUEUE can have up to 16 sort orders that exist in memory *simultaneously*.

For the purpose of this topic, sort orders not defined as the current active sort, but exist in memory by prior QUEUE actions (described later), are defined as a **memory key**.

All memory keys share the same queue of records, but each one orders the records according to the key that it is based on.

At any point in time during the life of the queue, one of the memory keys is the "default". The default memory key is the one that a SORT has been performed on most recently. If no SORT has been executed, the default memory key can be either unsorted, or sorted by the ADD(key) or PUT(key) methods ("sorting as you go")

The unsorted key is often used for non-keyed operations (i.e., POINTER()).

FREE() removes all memory keys.

The memory key based on the last key used in the keyed ADD, GET, PUT, or a SORT statement is called the **active key**. It is considered active while this particular statement is executing. If the active key did not exist before the keyed statement, it is created by taking the initial sequence of records defined by the default sort order, and is resorted using the new memory key.

SORT makes the active key the **default**. For example, if the active key existed before the SORT, no resorting is performed, because it doesn't need to.

GET(Queue,Key) retrieves the first record using the active key's (Key parameter) sort order that matches current content of the queue buffer. If no records can be found, the buffer is not changed and value of next call to POINTER() is undefined.

GET(Queue,Pointer) retrieves record with the relative position equal to the Pointer position in the default memory key.

ADD and PUT work differently for default key and all other memory keys. All memory keys that are not the default are updated using their key values always, even for non-keyed operations. Hence, their current sort is always correct.

For the original sort order, the situation is a little more complex. Here, the traditional rules are in effect:

PUT(Queue):

Writes the record back to the same relative position in the original sort order as the GET or ADD retrieved it. If the original sort order was modified since the last GET or ADD, it is marked as unsorted.

PUT(Queue,Pointer):

If the passed *Pointer* is equal to relative position in the original sort order of the record that has been retrieved by GET or ADD, the statement is equivalent to PUT(queue). Otherwise, the record is removed from its old position in the original sort order and is added to the one specified by the new pointer value. If the original sort order becomes broken after that, it is marked as unsorted.

PUT(Queue,Key)

If *Key* is a key that the original sort order is based on, and the key value is not changed, PUT updates the record value in original sort order. If the key value is changed, the record's value is removed from its old position and added to a new one, based on the new key value. The original sort order always remains unbroken in this case.

If *Key* is not a key that the original sort order is based on, the original sort order is marked as unsorted, the record's entry is removed from its old position, and a new one added immediately before the first record it finds with the key value based on *Key*.

Because the search algorithm is based on the history of work with this queue and its memory keys, it is impossible to say where the new position be. Use the POINTER() function to return it.

ADD(Queue)

Equivalent to **ADD(Queue,RECORDS(Queue)+1)**

ADD(Queue,Pointer)

The record in the queue buffer is added to given relative position in the original sort order. If the original sort order becomes broken after that, it is marked as unsorted.

ADD(Queue,Key)

The record in the queue buffer is added immediately before first record in the original sort order that has greater key value, or, to the end of the sort order if records with greater key value are not found. If *Key* is a key that the original sort order is based on, this is the correct position and the default sort order remains unbroken. Otherwise, it is marked as unsorted. Similar to using PUT on another key value, the position of the added record is unknown if the default sort order is not based on the *Key*.

Example 1:

```

Q  QUEUE
A   LONG
B   LONG
  END

```

```

CODE
  FREE(Q)
  Q.A = 1
  Q.B = 5
  ADD(Q, Q.A)

```

There is only one sort order based on the **(Q.A)** key; it is the active memory key

```

SORT (Q, Q.A, Q.B)

```

Now, there are two sort orders that exist, based on a **(Q.A)** and **(Q.A,Q.B)** keys. The latter key is now the active memory key.

The order of records is now:

```

(Q.A):          (1, 5)

```

```

(Q.A,Q.B)     (1, 5)

```

If we now execute:

```

  Q.A = 1
  Q.B = 1
  ADD (Q, Q.A, Q.B)

```

The order of records is now:

```

(Q.A):          (1, 5) (1, 1)

```

```

(Q.A,Q.B):     (1, 1) (1, 5)

```

Executing the statement:

```

  GET(Q, Q.A)

```

Retrieves the **(1, 5)** record, because it is the first record matching the key value in the current queue buffer based on **(Q.A)**.

Example 2:

```

Q  QUEUE
A   LONG
B   LONG
  END

```

```

CODE

```

```

FREE(Q)
Q.A = 1
Q.B = 5
ADD (Q)

```

There is one original sort order; and it is the active key.

```

SORT (Q, Q.A, Q.B)

```

Now, there is one sort order here based on the **(Q.A,Q.B)** key. It is now the new active key.

The current order of records is:

```

(Q.A,Q.B)    (1, 5)

```

After executing the following:

```

Q.A = 1
Q.B = 1
ADD (Q, Q.A, Q.B)

```

The new order of records becomes:

```

(Q.A,Q.B)    (1, 1) (1, 5)

```

Executing:

```

GET (Q, Q.A)

```

The sort order based on **(Q.A)** does not exist. Hence, it is created as a new sequence of records in default order resorted with the **(Q.A)** key.

The memory keys after this GET:

```

(Q.A)        (1, 1) (1, 5)

```

```

(Q.A,Q.B)    (1, 1) (1, 5)

```

The GET retrieves the **(1, 1)** record because it is the first record with the keyvalue that matches the current queue buffer based on **(Q.A)**.

5 - Declaration Attributes

Variable and Entity Attributes

AUTO (uninitialized local variable)

AUTO

The **AUTO** attribute allows a variable, declared within a PROCEDURE, to be allocated uninitialized stack memory. Without the AUTO attribute, a numeric variable is initialized to zero and a string variable is initialized to all blanks when its memory is assigned at run-time.

The AUTO attribute is used when you do not need to rely on an initial blank or zero value because you intend to assign some other value to the variable. This saves a small amount of run-time memory by eliminating the internal code necessary to perform the automatic initialization for the variable.

Example:

```
SomeProc PROCEDURE
SaveCustID LONG,AUTO      !Non-initialized local variable
```

See Also:

Data Declarations and Memory Allocation

BINARY (memo contains binary data)

BINARY

The **BINARY** attribute (PROP: BINARY) of a MEMO or BLOB declaration specifies the MEMO or BLOB will receive data that is not just ASCII characters. This attribute is normally used to store graphic images for display in an IMAGE field on screen. OEM conversion is not applied to MEMO or BLOB fields with the BINARY attribute. Some file drivers (Clarion, Btrieve, xBase) assume that the data in a BINARY MEMO or BLOB field is zero-padded, while non-BINARY data is space-padded.

Example:

```
Names FILE,DRIVER('Clarion'),PRE(Nam)
NbrKey KEY(Nam:Number)
Picture MEMO(48000),BINARY      !Binary memo - 48,000 bytes
Rec      RECORD
Number   SHORT
        END
      END
```

See Also:

MEMO

BLOB

IMAGE

OEM

BINDABLE (set runtime expression string variables)**BINDABLE**

The **BINDABLE** attribute declares a GROUP, QUEUE, FILE, or VIEW whose constituent variables are all available for use in a runtime expression string. The contents of each variable's NAME attribute is the logical name used in the dynamic expression. If no NAME attribute is present, the label of the variable (including prefix) is used. Space is allocated in the .EXE for the names of all of the variables in the structure. This creates a larger program that uses more memory than it normally would. Therefore, the BINDABLE attribute should only be used when a large proportion of the constituent fields are going to be used.

The BIND(group) form of the BIND statement must still be used in the executable code before the individual fields in the QUEUE structure may be used.

Example:

```
Names    QUEUE,BINDABLE                !Bindable Record structure
Name     STRING(20)
FileName STRING(8),NAME('FName')      !Dynamic name: FName
Dot      STRING(1)                    !Dynamic name: Dot
Extension STRING(3),NAME('EXT')       !Dynamic name: EXT
      END
CODE
      BIND(Names)
```

```
Names    FILE,DRIVER('Clarion'),BINDABLE !Bindable Record structure
Record   RECORD
Name     STRING(20)
FileName STRING(8),NAME('FName')      !Dynamic name: FName
Dot      STRING(1)                    !Dynamic name: Dot
Extension STRING(3),NAME('EXT')       !Dynamic name: EXT
      END
      END
CODE
      OPEN(Names)
      BIND(Names)
```

```
FileNames GROUP,BINDABLE              !Bindable group
FileName  STRING(8),NAME('FILE')      !Dynamic name: FILE
Dot       STRING('.')                 !Dynamic name: Dot
Extension STRING(3),NAME('EXT')       !Dynamic name: EXT
      END
```

See Also: BIND, UNBIND, EVALUATE

CREATE (allow data file creation)**CREATE**

The **CREATE** attribute (PROP:CREATE) of a FILE declaration allows a disk file to be created by the CREATE statement from within the PROGRAM where the FILE is declared. This adds some overhead, as all the file information must be contained in the executable program.

Example:

```
Names FILE,DRIVER('Clarion'),CREATE      !Declare a file, allow create
Rec   RECORD
Name   STRING(20)
      END
      END
```

DIM (set array dimensions)

DIM(*dimension*,...,*dimension*)

DIM	Declares a variable as an array.
<i>dimension</i>	A positive numeric constant which specifies the number of elements in this <i>dimension</i> of the array.

The **DIM** attribute declares a variable as an array. The variable is repeated the number of times specified by the *dimension* parameters. Multi-dimensional arrays may be thought of as nested. Each *dimension* in the array has a corresponding subscript. Therefore, referencing a variable in a three dimensional array requires three subscripts. There is no limit to the number of dimensions; however, the total size of an array is unlimited. Zero or negative array elements are invalid.

Subscripts identify which element of the array is being referenced. A subscript list contains a subscript for each *dimension* of the array. Each subscript is separated by a comma and the entire list is enclosed in brackets ([]). A subscript may be a numeric constant, expression, or function. The entire array may be referenced by the label of the array without a subscript list.

A GROUP structure array is a special case. Each level of nesting adds subscripts to the GROUP. Data declared within the GROUP is referenced using standard Field Qualification syntax with each subscript specified at the GROUP level at which it is dimensioned.

Example:

```

Scr   GROUP           !Characters on a DOS text-mode screen
Row   GROUP,DIM(25)  !Twenty-five rows
Pos   GROUP,DIM(80)  !Two thousand positions
Attr  BYTE           !Attribute byte
Char  BYTE           !Character byte
      END            !Terminate the group structures
      END
      END
! In the group above:
!   Scr           is a 4,000 byte GROUP
!   Scr.Row       is a 4,000 byte GROUP
!   Scr.Row[1]    is a 160 byte GROUP
!   Scr.Row[1].Pos is a 160 byte GROUP
!   Scr.Row[1].Pos[1] is a 2 byte GROUP
!   Scr.Row[1].Pos[1].Attr is a single BYTE
!   Scr.Row[1].Pos[1].Char is a single BYTE

Month STRING(10),DIM(12) !Dimension the month to 12
CODE
CLEAR(Month)             !Assign blanks to the entire array
Month[1] = 'January'     !Load the months into the array
Month[2] = 'February'
Month[3] = 'March'

```

See Also:

MAXIMUM

Prototype Parameter Lists (Passing Arrays)

DLL (set variable defined externally in .DLL)

`DLL([flag])`

DLL Declares a variable, FILE, QUEUE, GROUP, or CLASS defined externally in a .DLL.

flag A numeric constant, equate, or Project system define which specifies the attribute as active or not. If the *flag* is zero, the attribute is not active, just as if it were not present. If the *flag* is any value other than zero, the attribute is active.

The **DLL** attribute specifies that the declaration (this may any variable declaration, or a FILE, QUEUE, GROUP, or CLASS structure) on which it is placed is defined in a .DLL. A declaration with DLL attribute must also have the EXTERNAL attribute. The DLL attribute is required for 32-bit applications because .DLLs are relocatable in a 32-bit flat address space, which requires one extra dereference by the compiler to address the variable. The DLL attribute is not valid on variables declared within FILE, QUEUE, CLASS, or GROUP structures.

The declarations in all libraries (or .EXEs) must be EXACTLY the same (with the appropriate addition of the EXTERNAL and DLL attributes). If they are not exactly the same, data corruption could occur. Any incompatibilities between libraries cannot be detected by the compiler or linker, therefore it is the programmer's responsibility to ensure that consistency is maintained.

When using EXTERNAL and DLL on declarations shared by .DLLs and .EXE, only one .DLL should define the variable, FILE, CLASS, or QUEUE without the EXTERNAL and DLL attributes. All the other .DLLs (and the .EXE) should declare the variable, FILE, CLASS, or QUEUE with the EXTERNAL and DLL attributes. This ensures that there is only one memory allocation for the variable, FILE, CLASS, or QUEUE and all the .DLLs and the .EXE will reference the same memory when referring to that variable, FILE, or QUEUE.

One suggested way of coding large systems using many .DLLs and/or .EXEs that share the same variables would have one .DLL containing the actual data definition that only contains FILE and global variable definitions that are shared among all (or most) of the .DLLs and .EXEs. This makes one central library in which the actual file definitions are maintained. This one central .DLL is linked into all .EXEs that use those common files. All other .DLLs and/or .EXEs in the system would declare the common variables with the EXTERNAL and DLL attributes.

There is a pre-defined flag used by the IDE Project System and the Application Generator.

`_ABCDIIMode_` Used by the ABC template chain on all CLASS definitions to indicate that the CLASS is declared in an external .DLL. A project DEFINE is used to toggle the DLL mode.

Example:

```
TotalCount LONG,EXTERNAL,DLL(dll_mode)    !A variable declared in an external .DLL
Cust      FILE,PRE(Cus),EXTERNAL(''),DLL(1) !File defined in PROGRAM module of a .DLL
CustKey   KEY(Cus:Name)
Record    RECORD
Name      STRING(20)
          END
          END

DLLQueue  QUEUE,PRE(Que),EXTERNAL,DLL(1) !A queue declared in an external .DLL
TotalCount LONG
          END

EditEntryClass CLASS(EditClass),TYPE,MODULE('ABEIP.CLW'),LINK('ABEIP.CLW',|
_ABCLinkMode_),DLL(_ABCDllMode_)
CreateControl  PROCEDURE,VIRTUAL,PROTECTED
              END
```

See Also:

EXTERNAL

DRIVER (specify data file type)

DRIVER(*filetype* [,*driver string*])

DRIVER	Specifies the file system the file uses.
<i>filetype</i>	A string constant containing the name of the file manager (Btrieve, Clarion, etc.).
<i>driver string</i>	A string constant or variable containing any additional instructions to the file driver. All the valid values for this parameter are listed in each file driver's documentation.

The **DRIVER** attribute (PROP:DRIVER) specifies which file driver is used to access the data file. DRIVER is a required attribute of all FILE declarations.

Clarion programs use file drivers for physical file access. A file driver acts as a translator between a Clarion program and the file system, eliminating different access commands for each file system. File drivers allow access to files from different file systems without changes in the Clarion syntax.

The specific implementation method of each Clarion file access command is dependent on the file driver. Some commands may not be available in a file driver due to limitations in the file system. Each file driver is documented in the *User's Guide*. Any unsupported file access commands, FILE declaration attributes, data types, and/or file system idiosyncracies are listed there.

If the file is declared without the THREAD attribute, the *driver string* variable (PROP:DriverString) used with the DRIVER attribute should also be declared without the THREAD attribute.

Example:

```
Names FILE,DRIVER('Clarion')      !Begin file declaration
Record RECORD
Name   STRING(20)
      END
      END
```

DUP (allow duplicate KEY entries)

DUP

The **DUP** attribute (PROP:DUP) of a KEY declaration allows multiple records with the same key values to occur in a FILE. If the DUP attribute is omitted, attempting to ADD or PUT records with duplicate key values will generate the "Creates Duplicate Key" error, and the record will not be written to the file. During sequential processing using the KEY, records with duplicate key values are accessed in the physical order their entries appear in the KEY. The GET and SET statements generally access the first record in a set of duplicates.

The DUP attribute is unnecessary on INDEX declarations because an INDEX always allows duplicate entries.

Example:

```
Names    FILE,DRIVER('Clarion'),PRE(Nam)
NameKey  KEY(Nam:Name),DUP          !Declare name key, allow duplicate names
NbrKey   KEY(Nam:Number)           !Declare number key, no duplicates allowed
Rec      RECORD
Name     STRING(20)
Number   SHORT
        END
        END
```

See Also:

KEY

GET

SET

ENCRYPT (encrypt data file)

ENCRYPT

The **ENCRYPT** attribute (PROP:ENCRYPT) is used in conjunction with the OWNER attribute to disguise the information in a data file. ENCRYPT is only valid with an OWNER attribute. Even with a "hex-dump" utility, the data in an encrypted file is extremely difficult to decipher.

Example:

```
Names FILE,DRIVER('Clarion'),OWNER('Clarion'),ENCRYPT
Record RECORD
Name STRING(20)
      END
      END
```

See Also:

OWNER

EXTERNAL

EXTERNAL (set defined externally)**EXTERNAL**(*member*)

EXTERNAL	Specifies the variable, FILE, QUEUE, GROUP, or CLASS is defined in an external library.
<i>member</i>	A string constant (valid only on FILE, GROUP, or QUEUE declarations) containing the filename (without extension) of the MEMBER module containing the actual FILE definition (the one without an EXTERNAL attribute). If the FILE, GROUP or QUEUE is defined in a PROGRAM module or in a "universal member module" (i.e., MEMBER statement for that module has no parameter), an empty string ("") can be used as a parameter of the EXTERNAL attribute, or the parameter can be omitted.

The **EXTERNAL** attribute specifies the variable, FILE, QUEUE, GROUP, or CLASS on which it is placed is defined in an external library. Therefore, a variable, FILE, QUEUE, GROUP, or CLASS with the EXTERNAL attribute is declared and may be referenced in the Clarion code, but is not allocated memory--the memory for the variable, FILE, QUEUE, GROUP, or CLASS is allocated by the external library. This allows the Clarion program access to any variable, FILE, QUEUE, GROUP, or CLASS declared as public in external libraries. The EXTERNAL attribute is not valid on variables declared inside FILE, QUEUE, GROUP, or CLASS structures.

When using EXTERNAL(*member*) to declare a FILE shared by multiple libraries (.LIBs, or .DLLs and .EXE), only one library should define the FILE without the EXTERNAL attribute. All the other libraries (and the .EXE) should declare the FILE with the EXTERNAL attribute. This ensures that there is only one record buffer allocated for the FILE and all the libraries and the .EXE will reference the same memory when referring to data elements from that FILE.

The declarations in all libraries (or .EXEs) must be EXACTLY the same (with the appropriate addition of the EXTERNAL and DLL attributes). For example, the FILE declarations in all libraries (or .EXEs) that reference common files must contain exactly the same keys, memos, and fields declared in exactly the same order. If they are not exactly the same, data corruption could occur. Any incompatibilities between libraries cannot be detected by the compiler or linker, therefore it is the programmer's responsibility to ensure that consistency is maintained.

Do not place the OWNER, ENCRYPT, or NAME attributes on a FILE which has the EXTERNAL attribute. These attributes should only be on the FILE structure declared without the EXTERNAL, because the EXTERNAL declaration is actually a re-declaration of a FILE already declared elsewhere. Therefore, these attributes are unnecessary.

One suggested way of coding large systems using many .DLLs and/or .EXEs that share the same files would have one .DLL containing the actual FILE definition that only contains FILE and global variable definitions that are shared among all (or most) of the .DLLs and .EXEs. This makes one central library in which the actual file definitions are maintained. This one central .DLL is linked into all .EXEs that use those common files. All other .DLLs and/or .EXEs in the system would declare the common FILEs with the EXTERNAL attribute.

Example:

```

PROGRAM
MAP
  MODULE('LIB.LIB')
AddCount PROCEDURE                                !External library procedure
  END
END

TotalCount LONG,EXTERNAL                          !A variable declared in an external library
Cust  FILE,PRE(Cus),EXTERNAL('')                 !A File defined in a PROGRAM module
CustKey KEY(Cus:Name)                             ! whose .LIB is linked into this program
Record RECORD
Name  STRING(20)
      END
      END

Contact  FILE,PRE(Con),EXTERNAL('LIB01') !A File defined in a MEMBER module
ContactKey KEY(Con:Name)                    ! whose .LIB is linked into this program
Record RECORD
Name  STRING(20)
      END
      END

! The LIB.CLW file contains:
PROGRAM
MAP
  MODULE('LIB01')
AddCount PROCEDURE                                !Library procedure
  END
END

TotalCount LONG                                  !The TotalCount variable definition
Cust  FILE,PRE(Cus)                              !The Cust File definition where the
CustKey KEY(Cus:Name)                            ! record buffer is allocated
Record RECORD
Name  STRING(20)
      END
      END

```

```
CODE
!Executable code ...

! The LIB01.CLW file contains:
MEMBER('LIB')

Contact      FILE,PRE(Con)                !The Contact File definition where the
ContactKey   KEY(Con:Name)                ! record buffer is allocated
Record       RECORD
Name         STRING(20)
            END
            END

AddCount PROCEDURE
CODE
TotalCount += 1
```

FILTER (set view filter expression)**FILTER**(*expression*)**FILTER** Specifies a filter *expression* used to evaluate records to include in the VIEW.*expression* A string constant containing a logical expression.The **FILTER** attribute (PROP:FILTER) specifies a filter *expression* used to evaluate records to include in the VIEW.

The *expression* may reference any field in the VIEW, at all levels of JOIN structures. The entire *expression* must evaluate as true for a record to be included in the VIEW. The *expression* may contain any valid Clarion language logical expression. The *expression* is evaluated at runtime (just like the EVALUATE procedure), therefore you must BIND all variables used in the *expression*.

Use of MATCH with PROP:Filter and SQL Databases

The use of PROP:Filter as an SQL filter generator for SQL databases now supports converting the MATCH(*s1*, *s2*, *n*) function result to an appropriate SQL filter for all values of *n*, except Match:Regular. If you use the Match:Regular mode, the filter will be evaluated on the client side just like all other non-SQL convertible filters.

Other conversion rules:

MATCH(*s1*, *s2*, **Match:Simple**) will be converted to**s1 = s2**

for all SQL drivers.

MATCH(*s1*, *s2*, **Match:Soundex**) will be converted to**{fn SOUNDEX(*s1*)} = {fn SOUNDEX(*s2*)}**

for all ODBC back ends that support the SOUNDEX function.

MATCH(*s1*, *s2*, **Match:Soundex**) will be converted to**SOUNDEX(*s1*) = SOUNDEX(*s2*)**

for the Oracle Accelerator.

`MATCH(s1,s2,Match:Simple + Match:NoCase)` will be converted to
`{fn UPPER(s1)} = {fn UPPER(s2)}`

for all ODBC drivers.

`MATCH(s1,s2,Match:Simple + Match:NoCase)` will be converted to
`UPPER(s1) = UPPER(s2)`

for the Oracle Accelerator.

`MATCH(s1,s2) !Match:Wild mode`

Will be converted to

`s1 LIKE %`
 if the s2 parameter is using an asterisk (*), or

`s1 LIKE _`
 if the s2 parameter is using a question mark (?) wild card.

This is valid for all SQL drivers.

Adding the *Match:NoCase* mode to the *Match:Wild* mode is converted to
`{fn UPPER(s1)} LIKE % OR {fn UPPER(s1)} LIKE _`
 for all ODBC drivers.

Adding the *Match:NoCase* mode to the *Match:Wild* mode is converted to
`UPPER(s1) LIKE % OR UPPER(s1) LIKE _`
 for the Oracle Accelerator.

Example:

```
BRW1::View:Browse VIEW(Members)
    PROJECT(Mem:MemberCode,Mem:LastName,Mem:FirstName)
    END
KeyValue STRING(20)

!Get only orders for customer 9999 since order number 100
ViewOrder VIEW(Customer),FILTER('Cus:AcctNumber = 9999 AND Hea:OrderNumber > 100')
```

```

PROJECT(Cus:AcctNumber,Cus:Name)
JOIN(Hea:AcctKey,Cus:AcctNumber)      !Join Header file
PROJECT(Hea:OrderNumber)
JOIN(Dtl:OrderKey,Hea:OrderNumber)    !Join Detail file
PROJECT(Det:Item,Det:Quantity)
JOIN(Pro:ItemKey,Dtl:Item)           !Join Product file
PROJECT(Pro:Description,Pro:Price)
END
END
END
END !view

```

CODE

```

BIND('KeyValue',KeyValue)
BIND(Mem:Record)
KeyValue = 'Smith'
BRW1::View:Browse{PROP:Filter} = 'Mem:LastName = KeyValue' !Specify filter condition
OPEN(BRW1::View:Browse)                                     !Open the view
SET(BRW1::View:Browse)                                     !and set to the beginning
                                                           !of the filtered
                                                           !and ordered result set

```

CODE

```

OPEN((Customer,22h); OPEN((Header,22h); OPEN((Product,22h); OPEN(Detail,22h)
BIND('Cus:AcctNumber',Cus:AcctNumber)
BIND('Hea:OrderNumber',Hea:OrderNumber)
SET(Cus:AcctKey)
OPEN(ViewOrder)
LOOP
NEXT(ViewOrder)
IF ERRORCODE() THEN BREAK.
!Process the valid record
END
UNBIND('Cus:AcctNumber',Cus:AcctNumber)
UNBIND('Hea:AcctNumber',Hea:AcctNumber)
CLOSE(Header); CLOSE(Customer); CLOSE(Product); CLOSE(Detail)

```

See Also:

BIND

UNBIND

EVALUATE

IMPLEMENTS(add methods to a CLASS)**IMPLEMENTS**(*interface*)

IMPLEMENTS Adds additional methods to a CLASS.

interface A previously defined INTERFACE structure whose methods will be defined by the CLASS that is implementing the specified interface.

When a class IMPLEMENTS an interface, it inherits all methods that are defined in the INTERFACE. A class may IMPLEMENT multiple INTERFACES. The class must define all methods declared in each INTERFACE that is implemented.

Example:

```

MyInterface INTERFACE                    ! Interface structure
MyProc1        PROCEDURE                ! Method prototype
MyProc2        PROCEDURE                ! Method prototype
                                        END

MyClass CLASS, IMPLEMENTS(MyInterface) ! Class
                                        END

MyClass.MyInterface.MyProc1            !Method declaration
CODE

MyClass.MyInterface.MyProc2            !Method declaration
CODE

```

See Also:

INTERFACE

INTERFACE (class behavior definition)

```
label          INTERFACE ( [ parentinterface ] ) [, TYPE]
                [ methods ]
                END
```

INTERFACE A collection of methods to be used by the class that implements the interface.

parentinterface The label of a previously declared INTERFACE structure whose methods are inherited by the new INTERFACE. This may be an INTERFACE with the TYPE attribute.

TYPE Specify the INTERFACE is only a type definition. TYPE is implicit on an INTERFACE but may be explicitly specified.

methods PROCEDURE prototypes

An **INTERFACE** is a structure, which contains the *methods* (PROCEDURES) that define the behavior to be implemented by a CLASS. It cannot contain any property declarations. All methods defined within the INTERFACE are implicitly virtual. A period or the END statement must terminate an INTERFACE structure.

Derived INTERFACES (Inheritance)

An INTERFACE declared with the *parentinterface* parameter creates a *derived interface* that inherits all the *methods* of the named *parentinterface*. The *derived interface* may also contain its own *methods*.

Any *method* prototyped in the *derived interface* with the same name as a *method* in the *parentinterface* overrides the inherited *method* if both have the same parameter lists. If the two *methods* have different parameter lists, they create polymorphic functions in the *derived interface* that must follow the rules of Procedure Overloading.

VIRTUAL Methods (Polymorphism)

All methods in an INTERFACE are implicitly virtual, although the virtual attribute may be explicitly specified for clarity.

VIRTUAL *methods* in the *derived interface* may directly call the *parentinterface* method of the same name by prepending PARENT to the method's name. This allows incremental derivation wherein a *derived interface* method may simply call up to the *parentinterface* method to perform its functionality, and then extend it for the requirements of the *derived interface*.

Method Definition

The PROCEDURE definition of a *method* (its executable code, not its prototype) is defined by the CLASS that is implementing the INTERFACE. All methods for an interface must be defined in the IMPLEMENTING class.

Referencing INTERFACE methods in your code

You must call the *methods* of an INTERFACE by using dot notation syntax (by prepending the label of the CLASS to the label of the INTERFACE to the label of the *method*).

For example, using the following INTERFACE and CLASS declaration:

```
MyInterface INTERFACE
MyProc      PROCEDURE
            END
MyClass     CLASS, IMPLEMENTS(MyInterface)
            END
```

You may call the MyProc PROCEDURE as:

```
CODE
MyClass.MyInterface.MyProc
```

See Also:

IMPLEMENTS

INNER (set inner join operation)**INNER**

The **INNER** attribute (PROP:INNER) specifies the JOIN structure declares an "inner join" instead of the default "left outer join."

The VIEW structure defaults to a "left outer join," where all records for the VIEW's *primary file* are retrieved whether the secondary file named in the JOIN structure contains any related records or not. Specifying the INNER attribute on the JOIN creates an "inner join" so that only those *primary file* records with related secondary file records are retrieved. Inner joins are normally more efficient than outer joins.

PROP:INNER is an array property of a VIEW indicating the presence or absence of the INNER attribute on a specific JOIN. Each array element returns one ('1') if the JOIN has the INNER attribute and blank ("") if it does not. The JOINS are numbered within the VIEW starting with 1 as they appear within the VIEW structure. (READ ONLY)

Example:

```

AView VIEW(BaseFile)
  JOIN(ParentFile,'BaseFile.parentID = ParentFile.ID')      !JOIN 1
  JOIN(GrandParent.PrimaryKey, ParentFile.GrandParentID)  !JOIN 2
  END
  END
  JOIN(OtherParent.PrimaryKey,BaseFile.OtherParentID),INNER !JOIN 3
  END
  END

! AView{PROP:Inner,1} returns ''
! AView{PROP:Inner,2} returns ''
! AView{PROP:Inner,3} returns '1'

ViewOrder VIEW(Customer),ORDER('-Hea:OrderDate,Cus:Name')
  PROJECT(Cus:AcctNumber,Cus:Name,Cus:Zip)
  JOIN(Hea:AcctKey,Cus:AcctNumber),INNER      !Inner Join on Header
  PROJECT(Hea:OrderNumber,Hea:OrderDate)    ! gets only customers with orders
  JOIN(Dtl:OrderKey,Hea:OrderNumber),INNER  !Inner join on Detail file
  PROJECT(Det:Item,Det:Quantity)           ! is natural and more efficient
  JOIN(Pro:ItemKey,Dtl:Item),INNER         !Inner join on Product file
  PROJECT(Pro:Description,Pro:Price)       ! is natural and more efficient
  END
  END
  END
  END

```

See Also: JOIN

LINK (specify CLASS link into project)

LINK(*linkfile*, [*flag*])

LINK	Names a file to add to the link list for the current project.
<i>linkfile</i>	A string constant naming an file (without an extension .OBJ is assumed) to link into the project. Normally, this would be the same as the parameter to the MODULE attribute, but may explicitly name a .LIB or .OBJ file.
<i>flag</i>	A numeric constant, equate, or Project system define which specifies the attribute as active or not. If the <i>flag</i> is zero or omitted, the attribute is not active, just as if it were not present. If the <i>flag</i> is any value other than zero, the attribute is active.

A **LINK** attribute of a CLASS structure names a *linkfile* to add to the compiler's link list for the project. LINK is only valid on a CLASS structure.

Example:

```
OneClass CLASS,MODULE('OneClass'),LINK('OneClass',1)      !Link in OneClass.OBJ
LoadIt      PROCEDURE
ComputeIt   PROCEDURE
            END
```

See Also:

CLASS

MEMBER

MODULE

MODULE (specify CLASS member source file)**MODULE**(*sourcefile*)

MODULE	Names a MEMBER module or external library file.
<i>sourcefile</i>	A string constant. If the <i>sourcefile</i> contains Clarion language source code, this specifies the filename (extension is optional) of the source file that contains the PROCEDURES. If the <i>sourcefile</i> is an external library, this string may contain any unique identifier.

A **MODULE** attribute of a CLASS structure names a MEMBER module or external library file which contains the PROCEDURE definitions for the CLASS's member methods. MODULE is only valid on a CLASS structure.

Example:

```
OneClass CLASS,MODULE('OneClass')    !Method definitions in OneClass.CLW
LoadIt    PROCEDURE                  !LoadIt procedure prototype
ComputeIt PROCEDURE                  !ComputeIt procedure prototype
      END
```

See Also:

CLASS

MEMBER

LINK

PROCEDURE Prototypes

NAME (set external name)**NAME**([*name*])

NAME	Specifies an external name.
<i>name</i>	A string constant containing the external name or the label of a static string variable. This may be declared as Global data, Module data, or Local data with the STATIC attribute..

The **NAME** attribute (PROP:NAME) specifies an external name. The NAME attribute is completely independent of the EXTERNAL attribute--there is no required connection between the two, although both attributes may be used on the same declaration.

The NAME attribute may be placed on a PROCEDURE Prototype, FILE, KEY, INDEX, MEMO, any field declared within a FILE, any field declared within a QUEUE structure, or any variable not declared within a structure. The NAME attribute has different implications depending on where it is used.

PROCEDURE Prototype Usage

NAME may be specified on a PROCEDURE Prototype. The *name* supplies the external name used by the linker to identify the procedure or function from an external library.

Variable Usage

NAME may be used on any variable declared outside of any structure. This provides the linker with an external name to identify a variable declared in an external library. If the variable also has the EXTERNAL attribute, it is declared, and its memory is allocated, as a public variable in the external library. Without the EXTERNAL attribute, it is declared, and its memory is allocated, in the Clarion program, and it is declared as an external variable in the external library.

FILE Usage

On a FILE statement, NAME specifies the filename of the data file for the file driver. If the *name* does not contain a drive and path, the current drive and directory are assumed. If the extension is omitted, the directory entry assumes the file driver's default value.

Some file drivers require that KEYs, INDEXes, or MEMOs be in separate files. Therefore, a NAME may also be placed on a KEY, INDEX, or MEMO declaration. A NAME attribute without a *name* parameter defaults to the label of the declaration statement on which it is placed (including any specified prefix).

NAME may be used on any field declared within a RECORD structure (the *name* parameter must be a constant, in this case). This provides the file driver with the name of a field as it may be used in that driver's file system.

You can dynamically change the name of a field within a FILE using PROP:NAME as an array. The array element number references the ordinal position of the field within the FILE.

The NAME attribute is considered a part of the file structure, and care should be taken not to have threaded variables used by NAME referenced by non-threaded files.

For example:

```

FileName STRING(255)
TFileName STRING(255),THREAD

File FILE,NAME(TFileName)           !Wrong
File FILE,NAME(FileName)           !Valid
File FILE,NAME(TFileName),THREAD   !Valid
File FILE,NAME(FileName),THREAD    !Valid

```

QUEUE Usage

The NAME attribute on a variable declared in a QUEUE structure specifies an external *name* for queue processing. The *name* provides an alternate method of addressing the variables in the QUEUE which may be used by the SORT, GET, PUT, and ADD statements.

Example:

```

PROGRAM
MAP
MODULE('External.Obj')
AddCount PROCEDURE(LONG),LONG,C,NAME('_AddCount')  !C function named '_AddCount'
END
END

Cust FILE,PRE(Cus),NAME(CustName)                 !Filename in CustName variable
CustKey KEY('Name'),NAME('c:\data\cust.idx')      !Declare key, cust.idx
Record RECORD
Name STRING(20)                                   !Default NAME to 'Cus:Name'
END
END

SortQue QUEUE
Field1 STRING(10),NAME('FirstField')              !QUEUE SORT NAME
Field2 LONG,NAME('SecondField')                  !QUEUE SORT NAME
END

CurrentCnt LONG,EXTERNAL,NAME('Cur')             !Field declared public in
! external library as 'Cur'
TotalCnt LONG,NAME('Tot')                         !Field declared external
! in external library as 'Tot'

```

CODE

OPEN(Cust)

Cust{PROP:NAME,1} = 'Fred'

!Cus:Name field now referenced as 'Fred'

See Also:

PROCEDURE Prototypes, QUEUE, SORT, GET. PUT. ADD. FILE, KEY, INDEX, EXTERNAL

NOCASE (case insensitive KEY or INDEX)

NOCASE

The **NOCASE** attribute (PROP:NOCASE) of a KEY or INDEX declaration makes the sorted sequence of alphabetic characters insensitive to the ASCII upper/lower case sorting convention. All alphabetic characters in key fields are converted to upper case as they are written to the KEY. This case conversion has no effect on the case of the stored data. The NOCASE attribute has no effect on non-alphabetic characters.

Example:

```
Names    FILE,DRIVER('Clarion'),PRE(Nam)
NameKey  KEY(Nam:Name),NOCASE    !Declare name key, make case insensitive
NbrKey   KEY(Nam:Number)        !Declare number key
Rec      RECORD
Name     STRING(20)
Number   SHORT
        END
        END
```

See Also:

INDEX

KEY

OEM (set international string support)**OEM**

The **OEM** attribute (PROP:OEM) specifies that the FILE on which it is placed contains non-English language string data that was stored by a DOS based program or needs to be read by a DOS based program. These strings are automatically translated from the OEM ASCII character set data contained in the file to the ANSI character set for display in Windows. All string data in the record is automatically translated from the ANSI character set to the OEM ASCII character set before the record is written to disk.

The specific OEM ASCII character set used for the translation comes from the DOS code page loaded by the *country.SYS* file. This makes the data file specific to the language used for that code page, and means the data may not be useable on a computer with a different code page loaded. This attribute may not be supported by all file systems; consult the specific file driver's documentation.

Example:

```

Cust  FILE,DRIVER('TopSpeed'),PRE(Cus),OEM !Contains international strings
CustKey  KEY(Cus:Name)
Record  RECORD
Name    STRING(20)
        END
        END
Screen WINDOW('Window')
        ENTRY(@S20),USE(Cus:Name)
        BUTTON('&OK'),USE(?Ok),DEFAULT
        BUTTON('&Cancel'),USE(?Cancel)
        END
CODE
OPEN(Cust)           !Open Cust file
SET(Cust); NEXT(Cust) !Get record, ASCII strings are automatically
                    ! translated to ANSI character set
OPEN(Screen)        !Open window and display ANSI data
ACCEPT
CASE FIELD()
OF ?Ok
CASE EVENT()
OF EVENT:Accepted
  PUT(Cust)         !Put record, ANSI strings are automatically
                    ! translated to the OEM ASCII character set
                    ! per the loaded DOS code page

  BREAK
END
END
END
CLOSE(Screen);CLOSE(Cust)

```

OPT (exclude null KEY or INDEX entries)

OPT

The **OPT** attribute (PROP:OPT) excludes entries in the KEY or INDEX for records with "null" values in all fields comprising the KEY or INDEX. For the purpose of this attribute, a "null" value is defined as zero in a numeric field or all blank spaces (20h) in a string field.

Example:

```
Names  FILE,DRIVER('Clarion'),PRE(Nam)  !Declare a file structure
NameKey KEY(Nam:Name),OPT                !Declare name key, exclude blanks
NbrKey  KEY(Nam:Number),OPT              !Declare number key, exclude zeroes
Rec     RECORD
Name    STRING(20)
Number  SHORT
        END
        END
```

See Also:

INDEX

KEY

ORDER (set view sort order expression)**ORDER**(*expression list*)**ORDER** Specifies an *expression list* used to sort the records in the VIEW.*expression list* A single string constant containing one or more expressions. Each expression in the list must be separated by a comma from the preceding expression.

The **ORDER** attribute (PROP:ORDER) specifies an *expression list* used to sort the records in the VIEW. The expressions within the *expression list* evaluate from left to right, with the leftmost expression defining the most significant sort and the rightmost defining the least significant sort. Expressions that begin with a unary minus (-) sort in descending order.

The *expression* may reference any field in the VIEW, at all levels of JOIN structures. The expressions in the *expression list* may contain any valid Clarion language expression. The *expression list* is evaluated at runtime (just like the EVALUATE procedure), therefore you must BIND all variables used in the *expression*.

For non-SQL file systems, the VIEW will use keys to do most of the sorting wherever possible, sorting only groups of records which have the same key values, keeping one 'bucket' sorted. Therefore, additional sort fields on top of a key can be quite efficient.

For SQL file systems, PROP:SQLOrder is an SQL-only equivalent to PROP:ORDER. For both of these properties, if the first character of the expression assigned to them is a plus sign (+) then that expression is concatenated to the existing order expression. For PROP:SQLOrder, if the first character of the expression assigned is a minus sign (-) then the existing order expression is concatenated to that expression. If the first character is not plus (or minus), the new expression overrides the existing expression.

Example:

```
!Orders sorted in descending date order, then customer name (by name within each date)
ViewOrder VIEW(Customer),ORDER('-Hea:OrderDate,Cus:Name')
    PROJECT(Cus:AcctNumber,Cus:Name,Cus:Zip)
    JOIN(Hea:AcctKey,Cus:AcctNumber)           !Join Header file
    PROJECT(Hea:OrderNumber,Hea:OrderDate)
    JOIN(Dtl:OrderKey,Hea:OrderNumber)       !Join Detail file
    PROJECT(Det:Item,Det:Quantity)
    JOIN(Pro:ItemKey,Dtl:Item)               !Join Product file
    PROJECT(Pro:Description,Pro:Price)
    END
    END
    END
    END
CODE
ViewOrder{PROP:ORDER} = '-Hea:OrderDate,Pro:Price-Det:DiscountPrice'
```

!Orders sorted by greatest discount within descending order date

!ABC Implementation example of PROP:SQLORDER
!For browse, after files and window are opened

BRW1::View:Browse{PROP:SQLOrder} = 'Phone'

!where Phone is a column name in SQL database
!For reports, after files and progress window are opened:

Process:View{PROP:SQLOrder} = 'au_lname'

!where au_lname is a column name in SQL database

See Also:

BIND

UNBIND

EVALUATE

OVER (set shared memory location)**OVER**(*overvariable*)**OVER** Allows one memory address to be referenced two different ways.*overvariable* The label of a variable that already occupies the memory to be shared.

The **OVER** attribute allows one memory address to be referenced two different ways. The variable declared with the **OVER** attribute must not be larger than the *overvariable* it is being declared **OVER** (it may be smaller, though).

You may declare a variable **OVER** an *overvariable* which is part of the parameter list passed into a PROCEDURE.

A field within a GROUP structure cannot be declared **OVER** a *variable* outside that GROUP structure.

Example:

```
SomeProc PROCEDURE(PassedGroup)      !Proc receives a GROUP parameter

NewGroup GROUP,OVER(PassedGroup)     !Redeclare passed GROUP parameter
Field1  STRING(10)                   !Compiler warning issued that
Field2  STRING(2)                    !NewGroup must not be larger
      END                             !than PassedGroup

CustNote FILE,PRE(Csn)                !Declare CustNote file
Notes   MEMO(2000)                   !The memo field
Record  RECORD
CustID  LONG
      END
      END

CsnMemoRow STRING(10),DIM(200),OVER(Csn:Notes)
      !Csn:Notes memo may be addressed
      ! as a whole or in 10-byte chunks
```

See Also:

DIM

OWNER (declare password for data encryption)**OWNER**(*password*)**OWNER** Specifies a file encryption password.*password* A string constant or variable.

The **OWNER** attribute (PROP:OWNER) specifies the *password* which is used by the ENCRYPT attribute to encrypt the data. An "Invalid Data File" error occurs if the *password* does not match the password that was actually used to encrypt the file.

An OWNER attribute without an accompanying ENCRYPT attribute is allowed by some file systems.

If the file is declared without the THREAD attribute, the variable used with the OWNER attribute should also be declared without the THREAD attribute.

Example:

```
Customer FILE,DRIVER('Clarion'),OWNER('abCdeF'),ENCRYPT !Encrypt password "abCdeF"
Record RECORD
Name STRING(20)
END
END
```

See Also:

ENCRYPT

EXTERNAL


```
G2      LIKE(G1),PRE(Me2)          !Another GROUP LIKE the first containing same
CODE                                         !variables using the "Me2" prefix
IF Dtl:AcctNumber <> Mst:AcctNumber !Is it a new account
  Mem:Message = 'New Account'           !display message
  Me2:Message = 'Variable in LIKE group'
END
IF Detail.AcctNumber <> Masterfile.AcctNumber !Same expression
  G1.Message = 'New Account'           ! display message
  G2.Message = 'Same Variable in LIKE group'
END
```

See Also: Reserved Words, Field Qualification

PRIMARY (set relational primary key)

PRIMARY

The **PRIMARY** attribute (PROP:PRIMARY) specifies the KEY is unique, includes all records in the file, and does not allow "null" values in any of the fields comprising the KEY. This is the definition of a file's "Primary Key" per the relational database theory as expressed by E. F. Codd.

Example:

```
Names    FILE,DRIVER('TopSpeed'),PRE(Nam)    !Declare a file structure
NameKey  KEY(Nam:Name),OPT                    !Declare name key, exclude blanks
NbrKey   KEY(Nam:Number),PRIMARY             !Declare number key as the primary key
Rec      RECORD
Name     STRING(20)
Number   SHORT
        END
        END
```

See Also:

KEY

PRIVATE (set variable private to a CLASS module)**PRIVATE**

The **PRIVATE** attribute specifies that the variable on which it is placed is visible only to the PROCEDURES defined within the source module containing the methods of the CLASS structure (whether members of the CLASS or not). This encapsulates the data from other CLASSES.

PRIVATE is also valid when used with static (threaded or non-threaded) variables outside of a CLASS structure. If a static variable is declared with the PRIVATE attribute, the compiler generates it without a public external name. Hence, it can only be used by procedures defined in the same source module.

Example:

```

OneClass CLASS,MODULE('OneClass.CLW'),TYPE
PublicVar  LONG                !Declare a Public variable
PrivateVar LONG,PRIVATE        !Declare a Private variable
BaseProc  PROCEDURE(REAL Parm)!Declare a Public method
        END
TwoClass  OneClass             !Instance of OneClass
CODE
  TwoClass.PublicVar = 1       !Legal assignment
  TwoClass.PrivateVar = 1     !Illegal assignment

!OneClass.CLW contains:
MEMBER()
MAP
SomeLocalProc PROCEDURE
  END
OneClass.BaseProc PROCEDURE(REAL Parm)
CODE
  SELF.PrivateVar = Parm      !Legal assignment
SomeLocalProc PROCEDURE
CODE
  TwoClass.PrivateVar = 1     !Legal assignment

```

See Also:

CLASS

PROTECTED (set variable private to a CLASS or derived CLASS)**PROTECTED**

The **PROTECTED** attribute specifies that the variable on which it is placed is visible only to the PROCEDURES declared within the same CLASS structure (the methods of that CLASS) and any CLASS derived from the CLASS in which it is declared. This encapsulates the data from any code external to the specific CLASS and its derived CLASSES.

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

You could think of these as "semiprivate".

Example:

```

OneClass CLASS,MODULE('OneClass.CLW'),TYPE
PublicVar    LONG                      !Declare a Public variable
ProtectedVar LONG,PROTECTED            !Declare a Protected variable
BaseProc     PROCEDURE(REAL Parm)     !Declare a Public method
        END

TwoClass OneClass                      !Instance of OneClass
CODE
TwoClass.PublicVar = 1                  !Legal assignment
TwoClass.ProtectedVar = 1              !Legal assignment, illegal if PRIVATE

!OneClass.CLW contains:
MEMBER()
MAP
SomeLocalProc PROCEDURE
        END

OneClass.BaseProc PROCEDURE(REAL Parm)
CODE
SELF.ProtectedVar = Parm                !Legal assignment

SomeLocalProc PROCEDURE
CODE
TwoClass.ProtectedVar = 1                !Legal assignment

```

See Also:

CLASS

RECLAIM (reuse deleted record space)**RECLAIM**

The **RECLAIM** attribute (PROP:RECLAIM) specifies that the file driver adds new records to the file in the space previously used by a record that has been deleted, if available. Otherwise, the record is added at the end of the file. Implementation of RECLAIM is file driver specific and may not be supported in all file systems.

Example:

```
Names FILE,DRIVER('Clarion'),RECLAIM !Reuse deleted record space
Record RECORD
Name STRING(20)
      END
      END
```

STATIC (set allocate static memory)**STATIC**

The **STATIC** attribute specifies permanent memory allocation to a variable, **GROUP**, or data buffer of a **QUEUE**. If it is used with variables declared within a **PROCEDURE** or **ROUTINE**, they are allocated statically instead of using stack memory. **STATIC** makes values contained in the local variable or **QUEUE** data buffer "persistent" from one call to the procedure to the next. Static local variables that require initialization are initialized once only during the first call to the respective procedure or routine. Regarding a **QUEUE** structure, only the data buffer is allocated static memory -- **QUEUE** entries are always allocated memory dynamically on the heap.

Example:

```

SomeProc  PROCEDURE
SaveQueue QUEUE,STATIC          !Static QUEUE data buffer
Field1    LONG                  !Value retained between
Field2    STRING                !procedure calls
        END

AcctFile  STRING(64),STATIC      !STATIC needed for use as
        !variable in NAME attribute

Transactions FILE,DRIVER('Clarion'),PRE(TRA),NAME(AcctFile)
AccountKey KEY(TRA:Account),OPT,DUP
Record     RECORD
Account    SHORT                !Account code
Date       LONG                 !Transaction Date
Amount     DECIMAL(13,2)        !Transaction Amount
        END
        END

```

See Also:

Data Declarations and Memory Allocation

THREAD (set thread-specific memory allocation)

THREAD

The **THREAD** attribute declares a variable, FILE, GROUP, QUEUE, or CLASS which is allocated memory separately for each execution thread in the program. This makes the values dependent upon which thread is executing.

A threaded variable must be allocated static memory, so Local data with the THREAD attribute is automatically considered STATIC. This attribute creates runtime "overhead," particularly on Global or Module data. Therefore, it should be used only when necessary.

Variable and GROUP Usage

The THREAD attribute declares a static variable which is allocated memory separately for each execution thread in the program. This makes the value contained in the variable dependent upon which thread is executing. Whenever a new execution thread is begun, a new instance of the variable, specific to that thread, is created and initialized to blank or zero (unless the AUTO attribute is also present).

FILE Usage

The THREAD attribute (PROP:THREAD--valid only for a FILE) on a FILE declaration allocates memory for its record buffer, file control block, and other file structure elements separately for each execution thread as each thread is started. This makes the values contained in the record buffer and other file elements dependent upon which thread is executing. Memory for its record buffer, file control block, and other file elements is deallocated when the thread is closed.

Instances of a threaded FILE are considered independent FILEs. Therefore, a FILE must be OPENed and CLOSEd for each new instance.

QUEUE Usage

The THREAD attribute on a QUEUE declaration declares a static QUEUE data buffer which is allocated memory separately for each execution thread in the program. This makes the values contained in the QUEUE dependent upon which thread is executing. Whenever a new execution thread is begun, a new instance of the QUEUE, specific to that thread, is created.

Example:

```

PROGRAM
MAP
Thread1  PROCEDURE
Thread2  PROCEDURE
END

Names    FILE,DRIVER('Clarion'),PRE(Nam),THREAD  !Threaded file
NbrNdx   INDEX(Nam:Number),OPT
Rec      RECORD
Name     STRING(20)
Number   SHORT
        END
        END

GlobalVar LONG,THREAD  !Each execution thread gets its own copy OF GlobalVar

CODE
START(Thread1)
START(Thread2)

Thread1  PROCEDURE
LocalVar LONG,THREAD  !Local threaded variable (automatically STATIC)
CODE
OPEN(Names)           !OPEN creates new record buffer instance
SET(Names)            !containing the first record in the file
NEXT(Names)

Thread2  PROCEDURE
SaveQueue QUEUE,THREAD !Static QUEUE data buffer Thread-specific QUEUE
Name     STRING(20)
Number   SHORT
        END
        END

CODE
OPEN(Names)           !OPEN creates another new record buffer instance
SET(Names)            !containing the last record in the file
PREVIOUS(Names)

```

See Also:

START, Data Declarations and Memory Allocation, STATIC, AUTO

TYPE (type definition)

TYPE

The **TYPE** attribute creates a type definition for a GROUP, QUEUE, or CLASS (a "named structure"). The label of the named structure can then be used as a data type to define other similar GROUPs, QUEUEs, or CLASSes (or you can use LIKE). TYPE may also be used to define named structures passed to PROCEDUREs, allowing the receiving procedure to directly address components of the type definition using Field Qualification syntax.

A GROUP, QUEUE, or CLASS declaration with the TYPE attribute is not allocated any memory. While the data members of a CLASS with the TYPE attribute are not allocated memory, the methods prototyped in the CLASS must be defined for use by any subsequent objects declared as that type. EXTERNAL and DLL are irrelevant.

When a type definition is used to pass a named structure as a parameter to a PROCEDURE, the receiving procedure may directly address component fields in the passed QUEUE using the Field Qualification syntax. This is the preferred method of addressing the components of the passed structure.

There is also a legacy method of addressing the components of the passed structure. The named structure parameter declaration on the PROCEDURE definition statement (not the prototype) can instantiate a local prefix for the passed QUEUE as it names the passed QUEUE for the procedure. For example, PROCEDURE(LOC:PassedQueue) declares the procedure uses the LOC: prefix (along with the individual field names used in the type definition) to directly address component fields of the QUEUE passed as the parameter using the same type of syntax that the PRE attribute specifies. However, using Field Qualification syntax is preferable--locally instantiated prefixes are only maintained for backward compatibility.

Example:

```
MAP
MyProc1  PROCEDURE(PassQue) !Passes a QUEUE defined the same as PassGroup
END

PassQue  QUEUE,TYPE          !Type-definition for passed QUEUE parameters
First    STRING(20)          !first name
Middle   STRING(1)           !middle initial
Last     STRING(20)          !last name
END

NameQue  QUEUE(PassQue)      !Name queue-- same structure as PassQue
END                                             !End queue declaration

CODE
MyProc1(NameQue)              !Call proc passing NameQue as parameter

MyProc1  PROCEDURE(PassedQue)!Proc to receive QUEUE parameter
LocalVar STRING(20)
CODE
LocalVar = PassedQue.First    !Assign NameQue.First to LocalVar from parameter
```

See Also:

Field Qualification, Prototype Parameters Lists, CLASS, GROUP

6 - Windows

Window Structures

APPLICATION (declare an MDI frame window)

```

label  APPLICATION('title') [,AT( )] [,CENTER] [,SYSTEM] [,MAX] [,ICON( )] [,STATUS( )] [,HLP( )]
        [,CURSOR( )] [,TIMER( )] [,ALRT( )] [,ICONIZE] [,MAXIMIZE] [,MASK] [,FONT( )]
        [,MSG( )] [,IMM] [,AUTO] [,PALETTE()]
        [,WALLPAPER( )] [,TILED      | ] [, HSCROLL  | ] [, DOUBLE | ]
        | CENTERED | | VSCROLL  | | NOFRAME|
        | HVSCROLL | | RESIZE  |
    [ MENUBAR
        multiple menu and/or item declarations
    END ]
    [ TOOLBAR
        multiple control field declarations
    END ]
END

```

APPLICATION Declares a Multiple Document Interface (MDI) frame.

label A valid Clarion label (required).

title Specifies the title text for the application window (PROP:Text).

AT Specifies the initial size and location of the application window (PROP:AT). If omitted, default values are selected by the runtime library.

CENTER Specifies that the window's initial position is centered in the screen by default (PROP:CENTER). This attribute takes effect only if at least one parameter of the **AT** attribute is omitted.

SYSTEM Specifies the presence of a system menu (PROP:SYSTEM).

MAX Specifies the presence of a maximize control (PROP:MAX).

ICON Specifies the presence of a minimize control, and names a file or standard icon identifier for the icon displayed when the window is minimized (PROP:ICON).

STATUS	Specifies the presence of a status bar at the base of the application window (PROP:STATUS).
HLP	Specifies the "Help ID" associated with the APPLICATION window and provides the default for any child windows (PROP:HLP).
CURSOR	Specifies a mouse cursor to be displayed when the mouse is positioned over the APPLICATION window (PROP:CURSOR). If omitted, the Windows default cursor is used.
TIMER	Specifies periodic timed event generation (PROP:TIMER).
ALRT	Specifies "hot" keys active for the APPLICATION (PROP:ALRT).
ICONIZE	Specifies the APPLICATION is opened as an icon (PROP:ICONIZE).
MAXIMIZE	Specifies the APPLICATION is maximized when opened (PROP:MAXIMIZE).
MASK	Specifies pattern input editing mode of all ENTRY controls in the TOOLBAR (PROP:MASK).
FONT	Specifies the default font for all controls in the toolbar (PROP:FONT).
MSG	Specifies a string constant containing the default text to display in the status bar for all controls in the APPLICATION (PROP:MSG).
IMM	Specifies the window generates events whenever it is moved or resized (PROP:IMM).
AUTO	Specifies all toolbar controls' USE variables re-display on screen each time through the ACCEPT loop (PROP:AUTO).
PALETTE	Specifies the number of hardware colors used for graphics in the window (PROP:PALETTE).
WALLPAPER	Specifies the background image to display in the window's client area (PROP:WALLPAPER). The image stretches to fill the entire client area of the window unless the TILED or CENTERED attribute is also present.
TILED	Specifies the WALLPAPER image displays at its default size and is tiled to fill the entire client area of the window (PROP:TILED).
CENTERED	Specifies the WALLPAPER image displays at its default size and is centered in the entire client area of the window (PROP:CENTERED).
HSCROLL	Specifies a horizontal scroll bar is automatically added to the application frame when any portion of a child window lies horizontally outside the visible area (PROP:FSCROLL).
VSCROLL	Specifies a vertical scroll bar is automatically added to the application frame when any portion of a child window lies vertically outside the visible area (PROP:VSCROLL).
HVSCROLL	Specifies both vertical and horizontal scroll bars are automatically added to the application frame when any portion of a child window lies outside the visible area.

DOUBLE	Specifies a double-width frame around the window (PROP:DOUBLE).
NOFRAME	Specifies a window with no frame (PROP:NOFRAME).
RESIZE	Specifies a thick frame around the window which does allow window resizing (PROP:RESIZE).
MENUBAR	Defines the menu structure (optional). The menu specified in an APPLICATION is the "Global menu."
TOOLBAR	Defines a toolbar structure (optional). The toolbar specified in an APPLICATION is the "Global toolbar."

APPLICATION declares a Multiple Document Interface (MDI) frame window. MDI is a part of the standard Windows interface, and is used by Windows applications to present several "views" in different windows. This is a way of organizing and grouping these. The MDI frame window (APPLICATION structure) acts as a "parent" for all the MDI "child" windows (WINDOW structures with the MDI attribute). These MDI "child" windows are clipped to the APPLICATION frame and automatically moved when the frame is moved, and can be totally concealed by minimizing the parent.

There may be only one APPLICATION window open at any time in a Clarion Windows program, and it must be opened before any MDI "child" windows may be opened. However, non-MDI windows may be opened before or after the APPLICATION is opened, and may be on the same execution thread as the APPLICATION.

An MDI "child" window must not be on the same execution thread as the APPLICATION. Therefore, any MDI "child" window called directly from the APPLICATION must be in a separate procedure so the START procedure can be used to begin a new execution thread. Once started, multiple MDI "child" windows may be called in the new thread.

A "conventional" APPLICATION window would have the ICON, MAX, STATUS, RESIZE, and SYSTEM attributes. This creates an application frame window with minimize and maximize buttons, a status bar, a resizable frame, and a system menu. It would also have a MENUBAR structure containing the global menu items, and may have a TOOLBAR with "shortcuts" to global menu items. These attributes create a standard Windows look and feel for the application frame.

An APPLICATION window may not contain controls except within its MENUBAR and TOOLBAR structures, and cannot be used for any output. For output, document windows or dialog boxes are required (defined using the WINDOW structure).

When the APPLICATION window is first opened, it remains hidden until the first DISPLAY statement or ACCEPT loop is encountered. This enables any changes to be made to the appearance before it is displayed. Events for the APPLICATION window are processed by the first ACCEPT loop encountered after the APPLICATION window is first opened.

Events Generated:

EVENT:PreAlertKey	The user pressed an ALRT attribute hot key.
EVENT:AlertKey	The user pressed an ALRT attribute hot key.
EVENT:CloseWindow	The window is closing.
EVENT:CloseDown	The application is closing.
EVENT:OpenWindow	The window is opening.
EVENT:LoseFocus	The window is losing focus to another thread.
EVENT:GainFocus	The window is gaining focus from another thread.
EVENT:Suspend	The window still has input focus but is giving control to another thread to process timer events.
EVENT:Resume	The window still has input focus and is regaining control from an EVENT:Suspend.
EVENT:Timer	The TIMER attribute has triggered.
EVENT:Move	The user is moving the window. CYCLE aborts the move.
EVENT:Moved	The user has moved the window.
EVENT:Size	The user is resizing the window. CYCLE aborts the resize.
EVENT:Sized	The user has resized the window.
EVENT:Restore	The user is restoring the window's previous size. CYCLE aborts the resize.
EVENT:Restored	The user has restored the window's previous size.
EVENT:Maximize	The user is maximizing the window. CYCLE aborts the resize.
EVENT:Maximized	The user has maximized the window.
EVENT:Iconize	The user is minimizing the window. CYCLE aborts the resize.
EVENT:Iconized	The user has minimized the window.
EVENT:Completed	AcceptAll (non-stop) mode has finished processing all the window's controls.
EVENT:DDErequest	A client has requested a data item from this Clarion DDE server application.
EVENT:DDEadvise	A client has requested continuous updates of a data item from this Clarion DDE server application.
EVENT:DDEexecute	A client has executed a DDEEXECUTE statement to this Clarion DDE server application.
EVENT:DDEpoke	A client has sent unsolicited data to this Clarion DDE server application.

EVENT:DDEdata A DDE server has supplied an updated data item to this Clarion client application.

EVENT:DDEclosed A DDE server has terminated the DDE link to this Clarion client application.

Related Procedures:

ACCEPT, ALERT, EVENT, POST, REGISTER, UNREGISTER, YIELD, ACCEPTED, CHANGE, CHOICE, CLOSE, CONTENTS, CREATE, DESTROY, DISABLE, DISPLAY, ENABLE, ERASE, FIELD, FIRSTFIELD, FOCUS, GETFONT, GETPOSITION, HELP, HIDE, INCOMPLETE, LASTFIELD, MESSAGE, MOUSEX, MOUSEY, OPEN, POPUP, SELECT, SELECTED, SET3DLOOK, SETCURSOR, SETFONT, SETPOSITION, SETTARGET, UNHIDE, UPDATE

Example:

```
!An MDI application frame window with system menu, minimize and maximize
! buttons, a status bar, scroll bars, and a resizable frame, containing the
! main menu and toolbar for the application:
MainWin APPLICATION('My Application'),SYSTEM,MAX,ICON('MyIcon.ICO'),STATUS |
,HVSCROLL,RESIZE
MENUBAR
MENU('&File'),USE(?FileMenu)
ITEM('&Open...'),USE(?OpenFile)
ITEM('&Close'),USE(?CloseFile),DISABLE
ITEM('E&xit'),USE(?MainExit)
END
MENU('&Edit'),USE(?EditMenu)
ITEM('Cu&t'),USE(?CutText),KEY(CtrlX),STD(STD:Cut),DISABLE
ITEM('&Copy'),USE(?CopyText),KEY(CtrlC),STD(STD:Copy),DISABLE
ITEM('&Paste'),USE(?PasteText),KEY(CtrlV),STD(STD:Paste),DISABLE
END
MENU('&Window'),STD(STD:WindowList),LAST
ITEM('&Tile'),STD(STD:TileWindow)
ITEM('&Cascade'),STD(STD:CascadeWindow)
ITEM('&Arrange Icons'),STD(STD:ArrangeIcons)
END
MENU('&Help'),USE(?HelpMenu)
ITEM('&Contents'),USE(?HelpContents),STD(STD:HelpIndex)
ITEM('&Search...'),USE(?HelpSearch),STD(STD:HelpSearch)
ITEM('&How to Use Help'),USE(?HelpOnHelp),STD(STD:HelpOnHelp)
ITEM('&About MyApp...'),USE(?HelpAbout)
END
END
TOOLBAR
BUTTON('E&xit'),USE(?MainExitButton)
BUTTON('&Open'),USE(?OpenButton),ICON(ICON:Open)
END
```

```
    END
CODE
OPEN(MainWin)           !Open APPLICATION
ACCEPT                  !Display APPLICATION and accept user input
CASE ACCEPTED()        !Which control was chosen?
  OF ?OpenFile          !Open... menu selection
  OROF ?OpenButton      !Open button on toolbar
  START(OpenFileProc)  !Start new execution thread
  OF ?MainExit          !Exit menu selection
  OROF ?MainExitButton !Exit button on toolbar
  BREAK                !Break ACCEPT loop
  OF ?HelpAbout         !About... menu selection
  HelpAboutProc        !Call application information procedure
END
END
CLOSE(MainWin)         !Close APPLICATION
```

See Also:

ACCEPT

WINDOW

MDI

MENUBAR

TOOLBAR

WINDOW (declare a dialog window)

```

label  WINDOW('title') [,AT( )] [,CENTER] [,SYSTEM] [,MAX] [,ICON( )] [,STATUS( )] [,HLP( )]
        [,CURSOR( )] [,MDI] [,MODAL] [,MASK] [,FONT( )] [,GRAY] [,TIMER( )] [,ALRT( )]
        [,ICONIZE] [,MAXIMIZE] [,MSG( )] [,PALETTE( )] [,DROPID( )] [,IMM]
        [,AUTO] [,COLOR( )] [,TOOLBOX] [,DOCK( )] [,DOCKED( )]
        [,WALLPAPER( )] [, | TILED      | ] [, | HSCROLL  | ] [, | DOUBLE | ]
        | CENTERED | | | | VSCROLL  | | | NOFRAME |
        |           | | | | HVSCROLL  | | | RESIZE  |

[ MENUBAR
  menus and/or items
END ]

[ TOOLBAR
  controls
END ]

controls

END

```

WINDOW	Declares a document window or dialog box.
<i>label</i>	A valid Clarion label. A <i>label</i> is required.
<i>title</i>	A string constant containing the window's title text (PROP:Text).
AT	Specifies the initial size and location of the window (PROP:AT). If omitted, default values are selected by the runtime library.
CENTER	Specifies that the window's initial position is centered on screen relative to its parent window, by default (PROP:CENTER). This attribute takes effect only if at least one parameter of the AT attribute is omitted.
SYSTEM	Specifies the presence of a system menu (PROP:SYSTEM).
MAX	Specifies the presence of a maximize control (PROP:MAX).
ICON	Specifies the presence of a minimize control, and names a file or standard icon identifier for the icon displayed when the window is minimized (PROP:ICON).
STATUS	Specifies the presence of a status bar for the window (PROP:STATUS).

HLP	Specifies the "Help ID" associated with the window (PROP:HLP).
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the window (PROP:CURSOR). This cursor is inherited by the WINDOW's controls unless overridden on the individual control.
MDI	Specifies that the window conforms to normal MDI child-window behavior (PROP:MDI).
MODAL	Specifies the window is "system modal" and must be closed before the user may do anything else (PROP:MODAL).
MASK	Specifies pattern input editing mode of all entry controls in this window (PROP:MASK).
FONT	Specifies the default font for all controls in this window (PROP:FONT).
GRAY	Specifies that the window has a gray background for use with 3-D look controls (PROP:GRAY).
TIMER	Specifies periodic timed event generation (PROP:TIMER).
ALRT	Specifies "hot" keys active when the window has focus (PROP:ALRT).
ICONIZE	Specifies the window is opened as an icon (PROP:ICONIZE).
MAXIMIZE	Specifies the window is maximized when opened (PROP:MAXIMIZE).
MSG	Specifies a string constant containing the default text to display in the status bar for all controls in the window (PROP:MSG).
PALETTE	Specifies the number of hardware colors used for graphics in the window (PROP:PALETTE).
DROPID	Specifies the window may serve as a drop target for drag-and-drop actions (PROP:DROPID).
IMM	Specifies the window generates events whenever it is moved or resized (PROP:IMM).
AUTO	Specifies all window controls' USE variables re-display on screen each time through the ACCEPT loop (PROP:AUTO).
COLOR	Specifies a background color for the WINDOW and default background and selected colors for the controls in the WINDOW (PROP:COLOR).
TOOLBOX	Specifies the window is "always on top" and its controls never retain focus (PROP:TOOLBOX).
DOCK	Specifies a window with the TOOLBOX attribute is dockable (PROP:DOCK).
DOCKED	Specifies a window with the DOCK attribute is opens docked (PROP:DOCKED).
WALLPAPER	Specifies the background image to display in the window's client area (PROP:WALLPAPER). The image stretches to fill the entire client area of the window unless the TILED or CENTERED attribute is also present.

TILED	Specifies the WALLPAPER image displays at its default size and is tiled to fill the entire client area of the window (PROP:TILED).
CENTERED	Specifies the WALLPAPER image displays at its default size and is centered in the entire client area of the window (PROP:CENTERED).
HSCROLL	Specifies a horizontal scroll bar is automatically added to the window when any scrollable portion of the window lies horizontally outside the visible area (PROP:HSCROLL).
VSCROLL	Specifies a vertical scroll bar is automatically added to the window when any scrollable portion of the window lies vertically outside the visible area (PROP:VSCROLL).
HVSCROLL	Specifies both vertical and horizontal scroll bars are automatically added to the window when any scrollable portion of the window lies outside the visible area (PROP:HVSCROLL).
DOUBLE	Specifies a double-width frame around the window (PROP:DOUBLE).
NOFRAME	Specifies a window with no frame (PROP:NOFRAME).
RESIZE	Specifies a thick frame around the window, which does allow window resizing (PROP:RESIZE).
MENUBAR	Defines a menu structure (optional).

menus and/or items

MENU and/or ITEM declarations that define the menu selections.

TOOLBAR Defines a toolbar structure (optional).

controls

Control declarations that define tools available on the TOOLBAR, or the control fields in the WINDOW.

A **WINDOW** declares a document window or dialog box which may contain controls, and may be used to display output to the user. When the WINDOW is first opened, it remains hidden until the first DISPLAY statement or ACCEPT loop is encountered. This enables any changes to be made to the appearance before it is displayed. Any previously opened WINDOW on the same execution thread is disabled. Events for the WINDOW are processed by the first ACCEPT loop encountered after the WINDOW is first opened.

A WINDOW automatically receives a single-width border frame unless one of the DOUBLE, NOFRAME, or RESIZE attributes are specified. Screen coordinates are measured in dialog units. A dialog unit is defined as one-quarter the average character width and one-eighth the average character height of the font specified in the WINDOW's FONT attribute (or the system font, if no FONT attribute is specified on the WINDOW).

A WINDOW with the MODAL attribute is system modal; it takes exclusive control of the computer. This means that any other program running in the background halts its execution until the MODAL WINDOW is closed. Therefore, the MODAL attribute should be used only when absolutely necessary. Also, the RESIZE attribute is ignored, and the WINDOW cannot be moved when the MODAL attribute is present.

A WINDOW without the MDI attribute, when opened in an MDI program on an MDI execution thread, is application modal. This means that the user must respond before moving to any other window in the application. The user may, however, move to any other program running in Windows at the time. Non-MDI windows may be opened either before or after an APPLICATION is opened, and may be on the same execution thread as the APPLICATION or any MDI child window (application modal) or their own thread (not application modal).

A WINDOW with the MDI attribute is an MDI "child" window. MDI "child" windows are clipped to the APPLICATION frame and automatically moved when the frame is moved, and can be totally concealed by minimizing the parent APPLICATION. MDI "child" windows are modeless; the user may change to the top window of another execution thread, within the same application or any other application running in Windows, at any time. An MDI "child" window must not be on the same execution thread as the APPLICATION. Therefore, any MDI "child" window called directly from the APPLICATION must be in a separate procedure so the START procedure can be used to begin a new execution thread. Once started, multiple MDI "child" windows may be called in the new thread.

The MENUBAR specified in a WINDOW with the MDI attribute is automatically merged into the "Global menu" (from the APPLICATION) when the WINDOW receives focus unless either the WINDOW's or APPLICATION's MENUBAR has the NOMERGE attribute. A MENUBAR specified in a WINDOW without the MDI attribute is never merged into the "Global menu"--it always appears in the window itself.

The TOOLBAR specified in a WINDOW with the MDI attribute is automatically merged into the "Global toolbar" (from the APPLICATION) when the WINDOW receives focus, unless either the WINDOW's or APPLICATION's TOOLBAR has the NOMERGE attribute. The toolbar specified in a WINDOW without the MDI attribute is never merged into the "Global toolbar"--it always appears in the window itself.

A WINDOW with the TOOLBOX attribute is automatically "always on top" and its controls do not retain focus (just as if they all had the SKIP attribute). This creates a window whose controls all behave in the same manner as controls in the toolbar. Normally, a WINDOW with the TOOLBOX attribute would be executed in its own thread.

Events Generated:

EVENT:PreAlertKey	The user pressed an ALRT attribute hot key.
EVENT:AlertKey	The user pressed an ALRT attribute hot key.
EVENT:CloseWindow	The window is closing.
EVENT:CloseDown	The application is closing.
EVENT:OpenWindow	The window is opening.
EVENT:LoseFocus	The window is losing focus to another thread.
EVENT:GainFocus	The window is gaining focus from another thread.
EVENT:Suspend	The window still has input focus but is giving control to another thread to process timer events.
EVENT:Resume	The window still has input focus and is regaining control from an EVENT:Suspend.
EVENT:Docked	A TOOLBOX window has been docked.
EVENT:Undocked	A TOOLBOX window has been undocked.
EVENT:Timer	The TIMER attribute has triggered.
EVENT:Move	The user is moving the window. CYCLE aborts the move.
EVENT:Moved	The user has moved the window.
EVENT:Size	The user is resizing the window. CYCLE aborts the resize.
EVENT:Sized	The user has resized the window.
EVENT:Restore	The user is restoring the window's previous size. CYCLE aborts the resize.
EVENT:Restored	The user has restored the window's previous size.
EVENT:Maximize	The user is maximizing the window. CYCLE aborts the resize.
EVENT:Maximized	The user has maximized the window.
EVENT:Iconize	The user is minimizing the window. CYCLE aborts the resize.
EVENT:Iconized	The user has minimized the window.
EVENT:Completed	AcceptAll (non-stop) mode has finished processing all the window's controls.
EVENT:DDErequest	A client has requested a data item from this Clarion DDE server application.
EVENT:DDEadvise	A client has requested continuous updates of a data item from this Clarion DDE server application.
EVENT:DDEexecute	A client has executed a DDEEXECUTE statement to this Clarion DDE server application.

EVENT:DDEpoke	A client has sent unsolicited data to this Clarion DDE server application.
EVENT:DDEdata	A DDE server has supplied an updated data item to this Clarion client application.
EVENT:DDEclosed	A DDE server has terminated the DDE link to this Clarion client application.

Related Procedures:

ACCEPT, ALERT, EVENT, POST, REGISTER, UNREGISTER, YIELD, ACCEPTED, CHANGE, CHOICE, CLOSE, CONTENTS, CREATE, DESTROY, DISABLE, DISPLAY, ENABLE, ERASE, FIELD, FIRSTFIELD, FOCUS, GETFONT, GETPOSITION, HELP, HIDE, INCOMPLETE, LASTFIELD, MESSAGE, MOUSEX, MOUSEY, OPEN, POPUP, SELECT, SELECTED, SET3DLOOK, SETCURSOR, SETFONT, SETPOSITION, SETTARGET, UNHIDE, UPDATE

Example:

```

!MDI child window with system menu, minimize and maximize buttons, status bar,
! scroll bars, a resizable frame, with menu and toolbar which are merged into
the
!application's menubar and toolbar:
MDIChild WINDOW('Child One'),MDI,SYSTEM,MAX,ICON('Icon.ICO'),STATUS,HVSCROLL,RESIZE
  MENUBAR
    MENU('File'),USE(?FileMenu)
      ITEM('Close'),USE(?CloseFile)
    END
    MENU('Edit'),USE(?EditMenu)
      ITEM('Undo'),USE(?UndoText),KEY(CtrlZ),STD(STD:Undo)
      ITEM('Cu&t'),USE(?CutText),KEY(CtrlX),STD(STD:Cut)
      ITEM('Copy'),USE(?CopyText),KEY(CtrlC),STD(STD:Copy)
      ITEM('Paste'),USE(?PasteText),KEY(CtrlV),STD(STD:Paste)
    END
  END
  TOOLBAR
    BUTTON('Cut'),USE(?CutButton),ICON(ICON:Cut),STD(STD:Cut)
    BUTTON('Copy'),USE(?CopyButton),ICON(ICON:Copy),STD(STD:Copy)
    BUTTON('Paste'),USE(?PasteButton),ICON(ICON:Paste),STD(STD:Paste)
  END
  TEXT,HVSCROLL,USE(Pre:Field)
  BUTTON('&OK'),USE(?Exit),DEFAULT
END

!Non-MDI, system menu, maximize button, status bar, non-resizable frame,
NonMDI WINDOW('Dialog Window'),SYSTEM,MAX,STATUS
  TEXT,HVSCROLL,USE(Pre:Field),MSG('Enter some text here')
  BUTTON('&OK'),USE(?Exit),DEFAULT
END

!System-modal window with non-resizable frame, with only a message and Ok button:

```

```
ModalWin WINDOW('Modal Window'),MODAL
    IMAGE(ICON:Exclamation)
    STRING('An ERROR has occurred')
    BUTTON('&OK'),USE(?Exit),DEFAULT
END
```

See Also: ACCEPT, APPLICATION

MENUBAR (declare a pulldown menu)

```

MENUBAR [,USE( )] [, NOMERGE ]
    [ MENU( )
        [ ITEM( )
            [ MENU( )
                [ ITEM( )
                    END ]
            ]
        ]
    ]
    [ ITEM( )
        END
    ]

```

MENUBAR	Declares the menu for an APPLICATION or WINDOW.
USE	A field equate label to reference the menubar structure in executable code(PROP:USE).
NOMERGE	Specifies menu merging behavior.
MENU	A menu item with an associated drop box containing other menu selections.
ITEM	A menu item for selection.

The **MENUBAR** structure declares the pulldown menu selections displayed for an APPLICATION or WINDOW. MENUBAR must appear in the source code before any TOOLBAR or controls.

On an APPLICATION, the MENUBAR defines the Global menu selections for the program. These are active and available on all MDI "child" windows (unless the window's own MENUBAR structure has the NOMERGE attribute). If the NOMERGE attribute is specified on the APPLICATION's MENUBAR, then the menu is a local menu displayed only when no MDI child windows are open and there is no global menu.

On an MDI WINDOW, the MENUBAR defines menu selections that are automatically merged with the Global menu. Both the Global and the window's menu selections are then active while the MDI "child" window has input focus. Once the window loses focus, its specific menu selections are removed from the Global menu. If the NOMERGE attribute is specified on an MDI WINDOW's MENUBAR, the menu overwrites and replaces the Global menu.

On a non-MDI WINDOW, the MENUBAR is never merged with the Global menu. A MENUBAR on a non-MDI WINDOW always appears in the WINDOW, not on any APPLICATION which may have been previously opened.

Events generated by local menu items are sent to the WINDOW's ACCEPT loop in the normal way. Events generated by global menu items are sent to the active event loop of the thread which opened the APPLICATION (in a normal multi-thread application this means the APPLICATION's own ACCEPT loop).

Dynamic changes to menu items which reference the currently active window affect only the currently displayed menu, even if global items are changed. Changes made to the Global menu items when the APPLICATION is the current window, or which reference the global APPLICATION window affect the global portions of all menus, whether already open or not.

When a WINDOW's MENUBAR is merged into an APPLICATION's MENUBAR, the global menu selections appear first, followed by the local menu selections, unless the FIRST or LAST attributes are specified on individual menu selections.

A two-column drop menu can be achieved by assigning PROP:Max = 1 to the ITEM which should begin the second column.

Example:

```
!An MDI application frame window with main menu for the application:
MainWin APPLICATION('My Application')
  MENUBAR
    MENU('File'),USE(?FileMenu)
      ITEM('Open...'),USE(?OpenFile)
      ITEM('Close'),USE(?CloseFile),DISABLE
      ITEM('E&xit'),USE(?MainExit),LAST
    END
    MENU('Edit'),USE(?EditMenu)
      ITEM('Cu&t'),USE(?CutText),KEY(CtrlX),STD(STD:Cut),DISABLE
      ITEM('Copy'),USE(?CopyText),KEY(CtrlC),STD(STD:Copy),DISABLE
      ITEM('Paste'),USE(?PasteText),KEY(CtrlV),STD(STD:Paste),DISABLE
    END
    MENU('Window'),STD(STD:WindowList),LAST
      ITEM('Tile'),STD(STD:TileWindow)
      ITEM('Cascade'),STD(STD:CascadeWindow)
    END
    MENU('Help'),USE(?HelpMenu),LAST
      ITEM('Contents'),USE(?HelpContents),STD(STD:HelpIndex)
      ITEM('Search for Help On...'),USE(?HelpSearch),STD(STD:HelpSearch)
      ITEM('How to Use Help'),USE(?HelpOnHelp),STD(STD:HelpOnHelp)
      ITEM('About MyApp...'),USE(?HelpAbout)
    END
  END
END
```

```

!An MDI child window with menu for the window, merged into the
! application's menubar:
MDIChild WINDOW('Child One'),MDI
  MENUBAR
    MENU('File'),USE(?FileMenu)      !Merges into File menu
    ITEM('Pick...'),USE(?PickFile)   !Added to menu selections
  END
  MENU('Edit'),USE(?EditMenu)       !Merges into Edit menu
  ITEM('Undo'),USE(?UndoText),KEY(CtrlZ),STD(STD:Undo) !Added to menu
  END
  END
  TEXT,HVSCROLL,USE(Pre:Field)
  BUTTON('&OK'),USE(?Exit),DEFAULT
  END

!An MDI window with its own menu, overwriting the main menu:
MDIChild2 WINDOW('Dialog Window'),MDI,SYSTEM,MAX,STATUS
  MENUBAR,NOMERGE
    MENU('File'),USE(?FileMenu)
    ITEM('Close'),USE(?CloseFile)
  END
  MENU('Edit'),USE(?EditMenu)
    ITEM('Undo'),USE(?UndoText),KEY(CtrlZ),STD(STD:Undo)
    ITEM('Cu&t'),USE(?CutText),KEY(CtrlX),STD(STD:Cut)
    ITEM('Copy'),USE(?CopyText),KEY(CtrlC),STD(STD:Copy)
    ITEM('Paste'),USE(?PasteText),KEY(CtrlV),STD(STD:Paste)
  END
  END
  TEXT,HVSCROLL,USE(Pre:Field),MSG('Enter some text here')
  BUTTON('&OK'),USE(?Exit),DEFAULT
  END

```

```
!A non-MDI window with its own menu:
NonMDI WINDOW('Dialog Window'),SYSTEM,MAX,STATUS
  MENUBAR
  MENU('File'),USE(?FileMenu)
    ITEM('Close'),USE(?CloseFile)
  END
  MENU('Edit'),USE(?EditMenu)
    ITEM('Undo'),USE(?UndoText),KEY(CtrlZ),STD(STD:Undo)
    ITEM('Cu&t'),USE(?CutText),KEY(CtrlX),STD(STD:Cut)
    ITEM('Copy'),USE(?CopyText),KEY(CtrlC),STD(STD:Copy)
    ITEM('Paste'),USE(?PasteText),KEY(CtrlV),STD(STD:Paste)
  END
END
TEXT,HVSCROLL,USE(Pre:Field),MSG('Enter some text here')
BUTTON('&OK'),USE(?Exit),DEFAULT
END
```

TOOLBAR (declare a tool bar)

```
TOOLBAR [,AT( )] [,USE( )] [,CURSOR( )] [,FONT( )] [,NOMERGE] [,COLOR]
```

```
    [,WALLPAPER( )] [, | TILED      | ]
                    | CENTERED    | ]
```

controls

```
END
```

TOOLBAR	Declares tools for an APPLICATION or WINDOW.
AT	Specifies the initial size of the toolbar. If omitted, default values are selected by the runtime library.
USE	A field equate label to reference the toolbar in executable code (PROP:USE).
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the TOOLBAR. If omitted, the WINDOW or APPLICATION structure's CURSOR attribute is used, else the Windows default cursor is used.
FONT	Specifies the default display font for the controls in the TOOLBAR.
NOMERGE	Specifies tools merging behavior.
COLOR	Specifies a background color for the TOOLBAR and default background and selected colors for the controls in the TOOLBAR.
WALLPAPER	Specifies the background image to display in the toolbar (PROP:WALLPAPER). The image stretches to fill the entire toolbar unless the TILED or CENTERED attribute is also present.
TILED	Specifies the WALLPAPER image displays at its default size and is tiled to fill the entire toolbar (PROP:TILED).
CENTERED	Specifies the WALLPAPER image displays at its default size and is centered in the toolbar (PROP:CENTERED).
<i>controls</i>	Control field declarations that define the available tools.

The **TOOLBAR** structure declares the tools displayed for an APPLICATION or WINDOW. On an APPLICATION, the TOOLBAR defines the Global tools for the program. If the NOMERGE attribute is specified on the APPLICATION's TOOLBAR, the tools are local and are displayed only when no MDI child windows are open; there are no global tools. Global tools are active and available on all MDI "child" windows unless an MDI "child" window's TOOLBAR structure has the NOMERGE attribute. If so, the "child" window's tools overwrite the Global tools.

On an MDI WINDOW, the TOOLBAR defines tools that are automatically merged with the Global toolbar. Both the Global and the window's tools are then active while the MDI "child" window has input focus. Once the window loses focus, its specific tools are removed from the Global toolbar.

If the NOMERGE attribute is specified on an MDI WINDOW's TOOLBAR, the tools overwrite and replace the Global toolbar. On a non-MDI WINDOW, the TOOLBAR is never merged with the Global menu. A TOOLBAR on a non-MDI WINDOW always appears in the WINDOW, not on any APPLICATION which may have been previously opened.

Events generated by local tools are sent to the WINDOW's ACCEPT loop in the normal way. Events generated by global tools are sent to the active event loop of the thread which opened the APPLICATION. In a normal multi-thread application, this means the APPLICATION's own ACCEPT loop.

TOOLBAR controls generate events in the normal manner. However, they do not retain focus, and cannot be operated from the keyboard unless accelerator keys are provided. As soon as user interaction with a TOOLBAR control is done, focus returns to the window and local control which previously had it.

Dynamic changes to tools which reference the currently active window affect only the currently displayed toolbar, even if global tools are changed. Changes made to the Global toolbar when the APPLICATION is the current window, or which reference the global APPLICATION's window affect the global portions of all toolbars, whether already open or not. This means that, when an MDI child window is active, the APPLICATION frame's TOOLBAR controls displayed on the APPLICATION frame are actually copies of the frame's controls. This allows each MDI child to modify its own set of toolbar controls without affecting the controls displayed for other MDI child windows. The events for these controls are still processed by the APPLICATION's ACCEPT loop. For example, assuming a button declared in the APPLICATION's TOOLBAR has a field number of 150. The MDI Child window's procedure can modify the appearance of that button by directly setting the properties of control number 150, which would change its appearance only while the MDI Child window's procedure is active and has focus.

When a WINDOW's TOOLBAR is merged into an APPLICATION's TOOLBAR, the global tools appear first, followed by the local tools. The toolbars are merged so that the fields in the WINDOW's toolbar begin just right of the position specified by the value of the width parameter of the APPLICATION TOOLBAR's AT attribute. The height of the displayed toolbar is the maximum height of the "tallest" tool, whether global or local. If any part of a control falls below the bottom, the height is increased accordingly.

Example:

```

!An MDI application frame window containing the
! main menu and toolbar for the application:
MainWin APPLICATION('My Application'),SYSTEM,MAX,ICON('MyIcon.ICO'),STATUS |
,HVSCROLL,RESIZE
    MENUBAR
        ITEM('E&xit'),USE(?MainExit)
    END
    TOOLBAR
        BUTTON('Exit'),USE(?MainExitButton)
    END
END

!An MDI child window with toolbar for the window, merged into the
! application's toolbar:
MDIChild WINDOW('Child One'),MDI
    TOOLBAR
        BUTTON('Cut'),USE(?CutButton),STD(STD:Cut)
        BUTTON('Copy'),USE(?CopyButton),STD(STD:Copy)
        BUTTON('Paste'),USE(?PasteButton),STD(STD:Paste)
    END
    TEXT,HVSCROLL,USE(Pre:Field)
    BUTTON('&OK'),USE(?Exit),DEFAULT
END

!An MDI window with its own toolbar, overwriting the main toolbar:
MDIChild2 WINDOW('Dialog Window'),MDI,SYSTEM,MAX,STATUS
    TOOLBAR,NOMERGE
        BUTTON('Cut'),USE(?CutButton),STD(STD:Cut)
        BUTTON('Copy'),USE(?CopyButton),STD(STD:Copy)
        BUTTON('Paste'),USE(?PasteButton),STD(STD:Paste)
    END
    TEXT,HVSCROLL,USE(Pre:Field),MSG('Enter some text here')
    BUTTON('&OK'),USE(?Exit),DEFAULT
END

```

Window Overview

In most Windows programs there are three types of screen windows used: application windows, document windows, and dialog boxes. An application window is the first window opened in a Windows program, and it usually contains the main menu as the entry point to the rest of the program. All other windows in the program are document windows or dialog boxes.

Along with these three screen window types, there are two user interface design conventions that are used in Windows programs: the Single Document Interface (SDI), and the Multiple Document Interface (MDI).

An SDI program usually only contains linear logic that allows the user to take only one execution path (thread) at a time; it does not open separate execution threads which the user may move between. This is the same type of program logic used in most DOS programs. An SDI program would not contain a Clarion APPLICATION structure as its application window. The Clarion WINDOW structure (without an MDI attribute) is used to define an SDI program's application window, and the subsequent document windows or dialog boxes opened on top of it.

An MDI program allows the user to choose multiple execution paths (threads) and change from one to another at any time. This is a very common Windows program user interface. It is used by applications as a way of organizing and grouping windows which present several execution paths for the user to take.

A Clarion APPLICATION structure defines the MDI application window. The MDI application window acts as a parent for all the MDI child windows (document windows and dialog boxes), in that the child windows are clipped to its frame and automatically moved when the application frame is moved. They can also be concealed en masse by minimizing the parent. There may be only one APPLICATION open at any time in a Clarion Windows program.

Document windows and dialog boxes are very similar in that they are both defined as Clarion WINDOW structures. They differ in the conventional context in which they are commonly used and the conventions regarding appearance and attributes. In many cases, the difference is not distinguishable and does not matter. The generic term for both document windows and dialog boxes is "window" and that is the term used throughout this text.

Document windows usually display data. By convention they are movable and resizable. They usually have a title, a system menu, and maximize button. For example, in the Windows environment, the "Main" program group window that appears when you DOUBLE-CLICK on the "Main" icon in the Program Manager's desktop, is a document window.

Dialog boxes usually request information from the user or alert the user to some condition, usually prior to performing some action requested by the user. They may or may not be movable, and so, may or may not have a system menu and title. By convention, they are not resizable, although they can have a maximize button which gives the dialog two alternate sizes. A dialog box may be system modal (the user must respond before doing anything else in Windows), application modal (the user must respond before doing anything in the application), or modeless.

For example, in the Clarion environment, the window that appears from the File menu's Open selection is an application modal dialog box that requests the name of the file to open.

See Also:

Window Controls and Input Focus

Field Equate Labels

Window Controls and Input Focus

The objects placed in an APPLICATION or WINDOW structure are "controls." "Control" is a standard Windows term used to refer to any screen object--command buttons, text entry fields, radio buttons, list boxes, etc. In most DOS programs, the term "field" is usually used to refer to these objects. In this document, the terms "control" and "field" are generally interchangeable.

Controls appear only in MENUBARs, TOOLBARs, or WINDOW structures. Controls are available to the user to select and/or edit the data they contain only when it has "input focus." This occurs when the user uses the TAB key, the mouse, or an accelerator key combination to highlight the control.

A WINDOW also has "input focus" when it is the top WINDOW in the currently active execution thread. Since Clarion for Windows allows multi-threaded programs, the concept of which WINDOW currently has focus is important. Only the thread whose uppermost WINDOW has focus is active. The user may edit data in the WINDOW's control fields only when it has focus.

Field Equate Labels

Control Numbering

In WINDOW structures, every control (field) with a USE attribute is assigned a number by the compiler. By default, these field numbers begin with one (1) and are assigned to controls in the order they appear in the WINDOW structure code (the window itself is numbered zero). The actual assigned numbers can be overridden in the second parameter of the control's USE attribute.

The order of appearance in the WINDOW structure code determines the "natural" selection order of controls (which may be altered during program execution with the SELECT statement). The order of appearance in the WINDOW structure code is independent of the control's placement on the screen. Therefore, there is not necessarily any correlation between a control's position on screen and the field number assigned by the compiler.

In APPLICATION structures, every menu selection in the MENUBAR, and every control with a USE attribute placed in the TOOLBAR, is assigned a number by the compiler. By default, these numbers begin with negative one (-1) and are decremented by one (1) in the order the menu selections and controls appear in the APPLICATION structure code.

Equates for Control Numbers

There are a number of statements that use these compiler-assigned field numbers as parameters to indicate which controls are affected by the statement. It would be very tedious to "hard code" these numbers in order to use these statements. Therefore, Clarion provides a mechanism to address this problem: Field Equate Labels.

Field Equate Labels always begin with a question mark (?) followed by the label of the variable named in the control's USE attribute. The leading question mark indicates to the compiler a Field Equate Label. Field Equate Labels are very similar to normal EQUATE compiler directives. The compiler substitutes the field number for the Field Equate Label at compile time. This makes it unnecessary to know field numbers in advance.

Two or more controls with exactly the same USE variable in one WINDOW or APPLICATION structure would attempt to create the same Field Equate Label for all (each referencing a different field number). Therefore, when the compiler encounters this condition, all the Field Equate Labels for that USE variable are discarded. This makes it impossible to reference any of these controls in executable code, preventing confusion about which control you really want to reference. You can eliminate this problem by explicitly specifying the Field Equate Label for use by each control in the third parameter to the controls' USE attribute.

Array and Complex Structure Field Equates

Field Equate Labels for USE variables which are array elements always begin with a question mark (?) followed by the name of the USE variable followed by an underscore and the array element number. For example, the field equate for USE(ArrayField[1]) would be ?ArrayField_1. Multi-dimensioned arrays are treated similarly (?ArrayField_1_1, ?ArrayField_1_2, ...). You can override this default by explicitly specifying the Field Equate Label for use by each control in the third parameter to the controls' USE attribute.

Field Equate Labels for USE variables which are elements of a complex data structure always begin with a question mark (?) followed by the name of the USE variable with colons (:) replacing the periods (.). For example, the field equate for USE(Phones.Rec.Name) would be ?Phones:Rec:Name. This is done because Clarion labels may contain colons, but not periods, and a field equate is a label.

Using Field Equate Labels

Some controls' have USE attributes that can only be Field Equate Labels (a unique label with a leading question mark). This simply provides a way of referencing these fields in code or property assignment statements.

In executable code, there are many statements which use the field equate label to reference the control to affect (such as the DISPLAY statement). In all these statements, using a question mark (?) alone, without the USE variable name appended), always indicates performing the action on the current control that has input focus.

Example:

```
Window WINDOW('Dialog Window'),SYSTEM,MAX,STATUS
    TEXT,HVSCROLL,USE(Pre:Field)           !FEQ = ?Pre:Field
    ENTRY(@N3),HVSCROLL,USE(Pre:Array[1]) !FEQ = ?Pre:Array_1
    ENTRY(@N3),HVSCROLL,USE(File.MyField) !FEQ = ?File:MyField
    IMAGE(ICON:Exclamation),USE(?Image)   !USE attribute is a Field Equate Label
    BUTTON(' &OK '),USE(?Ok)              !USE attribute is a Field Equate Label
END

CODE
OPEN(Window)
?Ok{PROP:DEFAULT} = TRUE                  !Field Equates used in property assignments
?Image{PROP:Text} = 'MyImage.GIF'
ACCEPT
    DISPLAY(?)                             !Re-Display control with current input focus
END
```

Graphics Overview

Clarion supplies a set of "graphics primitives" procedures to allow drawing in windows and reports: ARC, BLANK, BOX, CHORD, ELLIPSE, IMAGE, LINE, PIE, POLYGON, ROUND BOX, SHOW, and TYPE. Controls always appear on top of any graphics drawn to the window. This means the graphics appear to underlay any controls in the window, so they don't get in the way of the controls the user needs to access.

Current Target

Graphics are always drawn to the "current target." Unless overridden with SETTARGET, the "current target" is the last window opened (and not yet closed) on the current execution thread and is the window with input focus. Drawings in a window are persistent--redraws are handled automatically by the runtime library.

Graphics in Reports

Graphics can also be drawn to a report. To do this, SETTARGET must first be used to nominate the REPORT as the "current target." Optionally, SETTARGET can nominate a specific report band to receive the graphics.

Consistent Graphics

Every window or report has its own current pen width, color, and style. Therefore, to consistently use the same pen (which does not use the default settings) across multiple windows, the SETPENWIDTH, SETPENCOLOR, and SETPENSTYLE statements should be issued for each window.

Graphics Coordinates

The graphics coordinate system starts with the x,y coordinates (0,0) at the top left corner of the window. The coordinates are specified in dialog units (unless overridden by the THOUS, MM, or POINTS attributes when used on graphics placed in a REPORT). A dialog unit is defined as one-quarter the average character width and one-eighth the average character height of the font specified in the window's FONT attribute (or the system font, if no FONT attribute is specified on the window).

Graphics drawn outside the currently visible portion of the window will appear if the window is scrolled. The size of the virtual screen over which the window may scroll automatically expands to include all graphics drawn to the window. Drawing graphics outside the visible portion of the window automatically causes the scroll bars to appear (if the window has the HSCROLL, VSCROLL, or HVSCROLL attribute).

7 - Reports

Report Structures

REPORT (declare a report structure)

```

label  REPORT([jobname]), AT( ) [, FONT( )] [, PRE( )] [, LANDSCAPE] [, PREVIEW] [, PAPER]
        [, COLOR( )] [ | THOUS      | ]
        | MM      |
        | POINTS  |
        [FORM
        controls
        END ]
        [HEADER
        controls
        END ]
label  DETAIL
        controls
        END
label  [BREAK( )
        group break structures
        END ]
        [FOOTER
        controls
        END ]
        END

```

REPORT	Declares the beginning of a report data structure.
<i>label</i>	The name by which the REPORT structure is addressed in executable code.
<i>jobname</i>	Names the print job for the Windows Print Manager (PROP:Text). If omitted, the REPORT's <i>label</i> is used.
AT	Specifies the size and location of the area for printing report detail, relative to the top left corner of the page (PROP:AT).
FONT	Specifies the default font for all controls in this report (PROP:FONT). If omitted, the printer's default font is used.
PRE	Specifies the label prefix for the report or structure.
LANDSCAPE	Specifies printing the report in landscape mode (PROP:LANDSCAPE). If omitted, printing defaults to portrait mode.
PREVIEW	Specifies report output to Windows metafiles; one file per report page (PROP:PREVIEW).
PAPER	Specifies the paper size for the report output. If omitted, the default printer's paper size is used.
COLOR	Specifies a background color for the REPORT and default background colors for the bands in the REPORT (PROP:COLOR).
THOUS	Specifies thousandths of an inch as the measurement unit used for all attributes which use coordinates (PROP:THOUS).
MM	Specifies millimeters as the measurement unit used for all attributes which use coordinates (PROP:MM).
POINTS	Specifies points as the measurement unit used for all attributes which use coordinates (PROP:POINTS). There are 72 points per inch, vertically and horizontally.
FORM	Page layout structure defining pre-printed items on every page.
<i>controls</i>	Report output controls.
HEADER	Page header structure, printed at the start of each page.
DETAIL	Report detail structure.
BREAK	A group break structure, defining the variable which causes a group break to occur when its value changes.
<i>group break structures</i>	Group break HEADER, FOOTER, and DETAIL structures, and/or other nested BREAK structures.
FOOTER	Page footer structure, printed at the end of each page.

The **REPORT** statement declares the beginning of a report data structure. A REPORT structure must terminate with a period or END statement. Within the REPORT, the FORM, HEADER, DETAIL, FOOTER, and BREAK structures are the components that format the output of the report. A REPORT must be explicitly opened with the OPEN statement.

A REPORT with the PREVIEW attribute sends the report output to Windows metafiles containing one report page per file. The PREVIEW attribute names a QUEUE to receive the names of the metafiles. You can then create a window to display the report in an IMAGE control, using the QUEUE field contents (the file names) to set the IMAGE control's {PROP:Text} property. This allows the end user to view the report before printing.

The REPORT's AT attribute defines the area of each page devoted to printing DETAIL structures. This includes any HEADERS and FOOTERS that are contained within a BREAK structure (group headers and footers).

Only DETAIL structures can (and must) be printed with the PRINT statement. All other report structures (HEADER, FOOTER, and FORM) automatically print at the appropriate place in the report.

The FORM structure prints on every page except pages containing DETAIL structures with the ALONE attribute. Its format is determined once at the beginning of the report. This makes it the logical place to design a pre-printed form template, which is filled in by the subsequent HEADER, DETAIL, and FOOTER structures. The page HEADER and FOOTER structures are not within a BREAK structure. They automatically print whenever a page break occurs.

The BREAK structure defines a group break. It may contain its own HEADER, FOOTER, and DETAIL structures, and/or other nested BREAK structures. It may also contain multiple DETAIL structures. The HEADER and FOOTER structures that are within a BREAK structure are the group header and footer. They are automatically printed when the value in a specified group break variable changes.

A REPORT data structure never defaults as the current target for runtime property assignment the way the most recently opened WINDOW or APPLICATION structure does. Therefore, the REPORT *label* must be explicitly named as the target, or the SETTARGET statement must be used to make the REPORT the current target, before using runtime property assignment to a REPORT control. Since the graphics commands draw graphics only to the current target, the SETTARGET statement must be used to make the REPORT the current target before using the graphics procedures on a REPORT.

Page-based Printing

Clarion reports use a page-based printing paradigm instead of the line-based paradigm used by some older report generators. Instead of printing each line as its values are generated, nothing is sent to the printer until an entire page is ready to print. This means that the "print engine" in the Clarion runtime library can do a lot of work for you, based on the attributes you specify in the REPORT structure.

Some of the things that the "print engine" in the Clarion runtime library does for you are:

- Prints "pre-printed" forms on each page, that are then filled in by the data
- Calculates totals (count, sum, average, minimum, maximum)
- Automatically handles page breaks, including page headers and footers
- Automatically handles group breaks, including group headers and footers
- Provides complete widow/orphan control.

This automatic functionality makes the executable code required to print a complex report very small, making your programming job easier. Since the "print engine" is page-based, the concepts of headers and footers lose their context indicating both page positioning and print sequence, and only retain their meaning of print sequence. Headers are printed at the beginning of a print sequence, and footers are printed at the end--their actual positioning on the page is irrelevant. For example, you could position the page footer, containing page totals, to print at the top of the page.

BREAK (declare group break structure)

label **BREAK**(*variable*) [,**USE**()] [,**NOCASE**]

group break structures

END

BREAK Declares a group break structure.

label The name by which the structure is addressed in executable code.

variable The variable whose change in value signals the group break (PROP:BreakVar).

USE A field equate label to reference the BREAK structure in executable code (PROP:USE).

NOCASE Specifies the check for a changed value in the *variable* is case insensitive.

group break structures

Group break HEADER, FOOTER, and DETAIL structures, and/or other nested BREAK structures.

The **BREAK** structure declares the *variable* which signals a group break when the value in the *variable* changes. A BREAK structure must be terminated with a period or END statement. It may contain its own HEADER, FOOTER, and DETAIL structures, and/or other nested BREAK structures. Only one HEADER and FOOTER are allowed in a BREAK structure; it may contain multiple DETAIL and/or BREAK structures.

The HEADER and FOOTER structures that are declared within a BREAK structure are the group header and footer. They automatically print surrounding equivalent values in the group break *variable*.

Example:

```
CustRpt REPORT                    !Declare customer report
Break1    BREAK(SomeVariable)
          HEADER                 !begin group header declaration
          !report controls
          END                     !end header declaration
GroupDet    DETAIL
          !report controls
          END                     !end detail declaration
          FOOTER                 ! begin group footer declaration
          !report controls
          END                     !end footer declaration
          END                     !end group break declaration
          END                     !End report declaration
```

DETAIL (report detail line structure)

```

label  DETAIL ,AT( ) [,FONT( )] [,ALONE] [,ABSOLUTE] [,PAGEBEFORE( )] [,PAGEAFTER( )]
        [,WITHPRIOR( )] [,WITHNEXT( )] [,USE( )] [,COLOR( )] [,TOGETHER]
        controls
        END

```

DETAIL	Declares items to be printed as the body of the report.
<i>label</i>	The name by which the structure is addressed in executable code.
AT	Specifies the offset and minimum width and height of the DETAIL, relative to the size of the area specified by the REPORT's AT attribute (PROP:AT).
FONT	Specifies the default font for all controls in this structure (PROP:FONT). If omitted, the REPORT's FONT attribute (if present) is used, or else the printer's default font is used.
ALONE	Declares the DETAIL structure must be printed on a page without FORM, (page) HEADER, or (page) FOOTER structures (PROP:ALONE).
ABSOLUTE	Declares the DETAIL prints at a fixed position relative to the page (PROP:ABSOLUTE).
PAGEBEFORE	Declares the DETAIL prints at the start of a new page, after activating normal page overflow actions (PROP:PAGEBEFORE).
PAGEAFTER	Declares the DETAIL prints, and then starts a new page by activating normal page overflow actions (PROP:PAGEAFTER).
WITHPRIOR	Declares the DETAIL prints on the same page as the DETAIL, or group HEADER or FOOTER that immediately precedes it during printing (PROP:WITHPRIOR).
WITHNEXT	Declares the DETAIL prints on the same page as the DETAIL, or group HEADER or FOOTER that immediately follows it during printing (PROP:WITHNEXT).
USE	A field equate label to reference the DETAIL structure in executable code (PROP:USE).
COLOR	Specifies a background color for the DETAIL and the default background color for the controls in the DETAIL (PROP:COLOR).
TOGETHER	Specifies that the DETAIL, or group HEADER or FOOTER structure (contained within a BREAK structure), is to only print on a single page, rather than being automatically split if there is not sufficient room for it (PROP:Together).
<i>controls</i>	Report output control fields.

The **DETAIL** structure declares items to be printed as the body of the report. A **DETAIL** structure must be terminated with a period or **END** statement. A **REPORT** may have multiple **DETAIL** structures.

A **DETAIL** structure is never automatically printed, therefore **DETAIL** structures are always explicitly printed by the **PRINT** statement. This means that a *label* is required for each **DETAIL** you wish to **PRINT**.

The **DETAIL** structure may be printed whenever necessary. Since you may have multiple **DETAIL** structures, they provide the ability to optionally print alternate print formats. This is determined by the logic in the executable code which prints the report.

DETAIL structures print within the detail print area specified by the **REPORT** statement's **AT** attribute. The **DETAIL** structure's **AT** attribute specifies the relative position, width and height of the detail to print. If there is horizontal room within the detail print area for multiple **DETAIL** structures, they print side-by-side.

Example:

```
CustRpt   REPORT           !Declare customer report
          HEADER           !begin page header declaration
          !structure elements
          END               !end header declaration
CustDetail1  DETAIL       !begin detail declaration
          !structure elements
          END               !end detail declaration
CustDetail2  DETAIL       !begin detail declaration
          !structure elements
          END               !end detail declaration
          END               !End report declaration

CODE
OPEN(CustRpt)
SET(SomeFile)
LOOP
  NEXT(SomeFile)
  IF ERRORCODE() THEN BREAK.
  IF SomeCondition
    PRINT(CustDetail1)
  ELSE
    PRINT(CustDetail2)
  END
END
END
CLOSE(CustRpt)
```

See Also:

PRINT, **AT**

FOOTER (page or group footer structure)

```
FOOTER ,AT( ) [,FONT( )] [,ABSOLUTE] [,PAGEBEFORE( )] [,PAGEAFTER( )]
      [,WITHPRIOR( )] [,WITHNEXT( )] [,ALONE] [,USE( )] [,COLOR( )] [,TOGETHER]
```

controls

END

FOOTER	Declares a page or group footer structure.
AT	Specifies the size and location of the FOOTER (PROP:AT).
FONT	Specifies the default font for all controls in this structure (PROP:FONT). If omitted, the REPORT's FONT attribute (if present) is used, or else the printer's default font is used.
ABSOLUTE	Declares the FOOTER prints at a fixed position relative to the page (PROP:ABSOLUTE). Valid only on a FOOTER within a BREAK structure.
PAGEBEFORE	Declares the FOOTER prints at the start of a new page, after activating normal page overflow actions (PROP:PAGEBEFORE). Valid only on a FOOTER within a BREAK structure.
PAGEAFTER	Declares the FOOTER prints, and then starts a new page by activating normal page overflow actions (PROP:PAGEAFTER). Valid only on a FOOTER within a BREAK structure.
WITHPRIOR	Declares the FOOTER prints on the same page as the DETAIL, group HEADER, or FOOTER that immediately precedes it during printing (PROP:WITHPRIOR). Valid only on a FOOTER within a BREAK structure.
WITHNEXT	Declares the FOOTER prints on the same page as the DETAIL, group HEADER, or FOOTER that immediately follows it during printing (PROP:WITHNEXT). Valid only on a FOOTER within a BREAK structure.
ALONE	Declares the (group) FOOTER structure must be printed on a page without FORM, (page) HEADER, or (page) FOOTER structures (PROP:ALONE).
USE	A field equate label to reference the FOOTER structure in executable code (PROP:USE).
COLOR	Specifies a background color for the FOOTER and the default background color for the controls in the FOOTER (PROP:COLOR).
TOGETHER	Specifies that the DETAIL, or group HEADER or FOOTER structure (contained within a BREAK structure), is to only print on a single page, rather than being automatically split if there is not sufficient room for it (PROP:Together).

controls Report output control fields.

The **FOOTER** structure declares the output which prints at the end of each page or group. A FOOTER structure must be terminated with a period or END statement.

A FOOTER structure that is not within a BREAK structure is a page footer. Only one page FOOTER is allowed in a REPORT. The page FOOTER is automatically printed whenever a page break occurs, at the page-relative position specified by its AT attribute.

The BREAK structure defines a group break. It may contain its own HEADER, FOOTER, and DETAIL structures, and/or other nested BREAK structures. It may also contain multiple DETAIL structures. The HEADER and FOOTER structures that are within a BREAK structure are the group header and footer. They are automatically printed when the value in a specified group break variable changes, at the next position available in the detail print area (specified by the REPORT's AT attribute). Only one FOOTER is allowed in a BREAK structure.

Example:

```
CustRpt  REPORT          !Declare customer report
        FOOTER          !begin page FOOTER declaration
        !report controls
        END              !end FOOTER declaration
Break1   BREAK(SomeVariable)
GroupDet  DETAIL
        !report controls
        END              !end detail declaration
        FOOTER          !begin group footer declaration
        !report controls
        END              !end footer declaration
        END              !end group break declaration
        END              !End report declaration
```

FORM (page layout structure)

```
FORM ,AT( ) [,FONT( )] [,USE( )] [,COLOR( )]
```

controls

```
END
```

FORM	Declares a report structure which prints on each page.
AT	Specifies the size and location, relative to the top left corner of the page, of the FORM (PROP:AT).
FONT	Specifies the default font for all controls in this report structure (PROP:FONT). If omitted, the REPORT's FONT attribute (if present) is used, or else the printer's default font is used.
USE	A field equate label to reference the FORM structure in executable code (PROP:USE).
COLOR	Specifies a background color for the FORM and the default background color for the controls in the FORM (PROP:COLOR).

controls Report output control fields.

FORM declares a report structure which prints on every page of the report (except pages containing DETAIL structures with the ALONE attribute). A FORM structure must be terminated with a period or END statement. Only one FORM is allowed in a REPORT structure. The FORM structure automatically prints during page overflow.

The printed output of the FORM is determined only once at the beginning of the report. The page positioning of the FORM does not affect the page positioning of any other report structure. Once printed, all other structures may "overwrite" the FORM. Therefore, FORM is most often used to design pre-printed forms which are filled in by the subsequent HEADER, DETAIL, and FOOTER structures. It may also be used to generate "watermarks" or page border graphics.

Example:

```
CustRpt REPORT      !Declare customer report
  FORM
    IMAGE( 'LOGO.BMP' ),AT(0,0,1200,1200),USE(?I1)
    STRING(@N3),AT(6000,500,500,500),PAGENO
  END
GroupDet DETAIL
  !report controls
  END
END                !End report declaration
```

HEADER (page or group header structure)

```

HEADER ,AT( ) [,FONT( )] [,ABSOLUTE] [,PAGEBEFORE( )] [,PAGEAFTER( )]
      [,WITHPRIOR( )] [,WITHNEXT( )] [,ALONE] [,USE( )] [,COLOR( )] [,TOGETHER]

```

controls

END

HEADER	Declares a page or group header structure.
AT	Specifies the size and location of the HEADER (PROP:AT).
FONT	Specifies the default font for all controls in this structure (PROP:FONT). If omitted, the REPORT's FONT attribute (if present) is used, or else the printer's default font is used.
ABSOLUTE	Declares the HEADER prints at a fixed position relative to the page (PROP:ABSOLUTE). Valid only on a HEADER within a BREAK structure.
PAGEBEFORE	Declares the HEADER prints at the start of a new page after activating normal page overflow actions (PROP:PAGEBEFORE). Valid only on a HEADER within a BREAK structure.
PAGEAFTER	Declares the HEADER prints, and then starts a new page by activating normal page overflow actions (PROP:PAGEAFTER). Valid only on a HEADER within a BREAK structure.
WITHPRIOR	Declares the HEADER prints on the same page as the DETAIL, group HEADER, or FOOTER that immediately precedes it during printing (PROP:WITHPRIOR). Valid only on a HEADER within a BREAK structure.
WITHNEXT	Declares the HEADER prints on the same page as the DETAIL, group HEADER, or FOOTER that immediately follows it during printing (PROP:WITHNEXT). Valid only on a HEADER within a BREAK structure.
ALONE	Declares the (group) HEADER structure must be printed on a page without FORM, (page) HEADER, or (page) FOOTER structures (PROP:ALONE).
USE	A field equate label to reference the HEADER structure in executable code (PROP:USE).
COLOR	Specifies a background color for the HEADER and the default background color for the controls in the HEADER (PROP:COLOR).
TOGETHER	Specifies that the DETAIL, or group HEADER or FOOTER structure (contained within a BREAK structure), is to only print on a single page, rather than being automatically split if there is not sufficient room for it (PROP:Together).

controls Report output control fields.

The **HEADER** structure declares the output which prints at the beginning of each page or group. A **HEADER** structure must be terminated with a period or **END** statement.

A **HEADER** structure that is not within a **BREAK** structure is a page header. Only one page **HEADER** is allowed in a **REPORT**. The page **HEADER** is automatically printed whenever a page break occurs, at the page-relative position specified by its **AT** attribute.

The **BREAK** structure defines a group break. It may contain its own **HEADER**, **FOOTER**, and **DETAIL** structures, and/or other nested **BREAK** structures. It may also contain multiple **DETAIL** structures. The **HEADER** and **FOOTER** structures that are within a **BREAK** structure are the group header and footer. They are automatically printed when the value in a specified group break variable changes, at the next position available in the detail print area (specified by the **REPORT**'s **AT** attribute). Only one **HEADER** is allowed in a **BREAK** structure.

Example:

```
CustRpt REPORT      !Declare customer report
      HEADER      ! begin page header declaration
      !report controls
      END          ! end header declaration
Break1  BREAK(SomeVariable)
      HEADER      ! begin group header declaration
      !report controls
      END          ! end header declaration
GroupDet  DETAIL
      !report controls
      END          ! end detail declaration
      END          ! end group break declaration
      END          !End report declaration
```

Printer Control Properties

These properties control report and printer behavior. All of these properties can be used with either the PRINTER built-in variable or the label of the report as the *target*, however they may not all make sense with both. These properties are contained in the PRNPROP.CLW file, which you must explicitly INCLUDE in your code in order to use them.

PROPPRINT:DevMode

The entire device mode (devmode) structure as defined in the Windows Software Development Kit. This provides direct API access to all printer properties. Consult a Windows API manual before using this.

The devmode structure is different in 32-bit (consult a Windows API manual). However, the following properties are the most common and useful:

DM_ORIENTATION	DM_PAPERSIZE	DM_PAPERLENGTH
DM_PAPERWIDTH	DM_SCALE	DM_COPIES
DM_DEFAULTSOURCE	DM_PRINTQUALITY	DM_POSITION
DM_DISPLAYORIENTATION	DM_DISPLAYFIXEDOUTPUT	DM_COLOR
DM_DUPLEX	DM_YRESOLUTION	DM_TTOPTION
DM_COLLATE	DM_FORMNAME	DM_LOGPIXELS
DM_BITSPERPEL	DM_PELSWIDTH	DM_PELSHEIGHT
DM_DISPLAYFLAGS	DM_NUP	DM_DISPLAYFREQUENCY
DM_ICMMETHOD	DM_ICMINTENT	DM_MEDIATYPE
DM_DITHERTYPE	DM_PANNINGWIDTH	DM_PANNINGHEIGHT

PROPPRINT:Collate

Specify the printer should collate the output: 0=off, 1=on (not supported by all printers).

PROPPRINT:Color

Color or monochrome print flag: 1=mono, 2=color (not supported by all printers).

PROPPRINT:Context

Returns the handle to the printer's device context after the first PRINT statement for the report, or an information context before the first PRINT statement. This may not be set for the built-in Global PRINTER variable and is normally only read (not set).

PROPPRINT:Copies

The number of copies to print (not supported by all printers).

PROPPRINT:Device

The name of the Printer as it appears in the Windows Printer Dialog. If multiple printer names start with the same characters, the first encountered is used (not case sensitive). May be set for the PRINTER built-in variable only before the report is open.

PROPPRINT:Driver

The printer driver's filename (without the .DLL extension).

PROPPRINT:Duplex

The duplex printing mode (not supported by all printers). Equates(DUPLEX::xxx) for the standard choices are listed in the PRNPROP.CLW file.

PROPPRINT:Extend

PROPPRINT:Extend can be set to TRUE or FALSE at runtime, and references the REPORT target. This determines whether or not the runtime library generates extra information inside the WMF files, which is being used to generate the new report output formats (XML, HTML, PDF etc.).

Set this property to FALSE (0) if you do not want alternate output formats and prefer smaller, compact WMF files (this is default behavior).

PROP:Extend is the runtime property that can be used to set the information for the EXTEND attribute.

PROPPRINT:FontMode

The TrueType font mode. Equates (FONTMODE:xxx) for the modes are listed in the PRNPROP.CLW file.

PROPPRINT:FromMin

When set for the built-in PRINTER variable, this forces the value into the "From:" page number in the PRINTERDIALOG. Specify -1 to disable ranges

PROPPRINT:FromPage

The page number on which to start printing. Specify -1 to print from the start.

PROPPRINT:Paper

Standard paper size. Equates (PAPER:xxx) for the standard sizes are listed in the PRNPROP.CLW file. This defines the dimensions of the .WMF files that are created by the Clarion runtime library's "print engine."

PROPPRINT:PaperBin

The paper source. Equates (PAPERBIN:xxx) for the standard locations are listed in the PRNPROP.CLW file.

PROPPRINT:PaperHeight

The paper height in tenths of millimeters (mm/10). There are 25.4 mm per inch. Used when setting PROPPRINT:Paper to PAPER:Custom (not normally used for laser printers).

PROPPRINT:PaperWidth

The paper width in tenths of millimeters (mm/10). There are 25.4 mm per inch. Used when setting PROPPRINT:Paper to PAPER:Custom (not normally used for laser printers).

PROPPRINT:Percent

The scaling factor used to enlarge or reduce the printed output, in percent (not supported by all printers). This defaults to 100 percent. Set this value to print at the desired percentage (if your printer and driver support scaling). For example, set to 200 to print at double size, or 50 to print at half size.

PROPPRINT:Port

Output port name (LPT1, COM1, etc.).

PROPPRINT:PrintToFile

The Print to File flag: 0=off, 1=on.

PROPPRINT:PrintToName

The output filename when printing to a file.

PROPPRINT:Resolution

The print resolution in Dots Per Inch (DPI). Equates (RESOLUTION:xxx) for the standard resolutions are listed in the PRNPROP.CLW file. Must be issued before the report is open.

PROPPRINT:ToMax

When set for the built-in PRINTER variable, this forces the value into the "To:" page number in the PRINTERDIALOG. Specify -1 to disable ranges

PROPPRINT:ToPage

The page number on which to end printing. Specify -1 to print to end.

PROPPRINT:Yresolution

Vertical print resolution in Dots Per Inch (DPI). Equates (RESOLUTION:xxx) for the standard resolutions are listed in the PRNPROP.CLW file.

Example:

```
SomeReport  REPORT
            END

CODE
PRINTER{PROPPRINT:Device} = 'Epson'           !Pick 1st Epson in the list
PRINTER{PROPPRINT:Port}   = 'LPT2:'         !Send report to LPT2
PRINTER{PROPPRINT:Percent} = 250           !page printed 2.5 times normal
PRINTER{PROPPRINT:Copies} = 3              !print 3 copies of each page
PRINTER{PROPPRINT:Collate} = False         !print 1,1,1,2,2,2,3,3,3,...
PRINTER{PROPPRINT:Collate} = True         !print 1,2,3..., 1,2,3...,
PRINTER{PROPPRINT:PrintToFile} = True     !print to a file
PRINTER{PROPPRINT:PrintToName} = 'OUTPUT.RPT' !filename to print to

OPEN(SomeReport)                               !Open report after setting PRINTER prope
SomeReport{PROPPRINT:Paper} = PAPER:User     !Custom paper size
SomeReport{PROPPRINT:PAPERHeight} = 6 * 254 !6" form height
SomeReport{PROPPRINT:PAPERWidth} = 3.5 * 254 !3.5" form width
```

Page Overflow

Page Overflow occurs when the PRINT statement cannot fit a DETAIL structure on a page. This may be due to a lack of space, or the presence of the PAGEBEFORE or PAGEAFTER attribute on a DETAIL structure. The following steps occur during page overflow, in this sequence:

- 1 If the REPORT has a page FOOTER, it prints at the position specified by its AT attribute.
- 2 The page counter is incremented.
- 3 If the REPORT has a FORM structure, it prints at the position specified by its AT attribute.
- 4 If the REPORT has a page HEADER, it prints at the position specified by its AT attribute.

Related Procedures: CLOSE, OPEN, ENDPAGE, PRINT

8 - Controls

Control Declarations

BOX (declare a box control)

BOX ,AT() [,USE()] [,DISABLE] [,COLOR()] [,FILL()] [,ROUND] [,FULL] [,SCROLL] [,HIDE]
[,LINEWIDTH()] [,LAYOUT()] [,EXTEND()]

BOX	Places a rectangular box on the window or report.
AT	Specifies the initial size and location of the control (PROP:AT). If omitted, default values are selected by the runtime library.
USE	A field equate label to reference the control in executable code (PROP:USE).
DISABLE	Specifies the control appears dimmed when the WINDOW (or APPLICATION) is first opened (PROP:DISABLE).
COLOR	Specifies the color for the border of the control (PROP:COLOR). If omitted, there is no border.
FILL	Specifies the fill color for the control (PROP:FILL). If omitted, the box is not filled with color.
ROUND	Specifies the box corners are rounded (PROP:ROUND). If omitted, the corners are square.
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter (PROP:FULL). Not valid in a REPORT.
SCROLL	Specifies the control scrolls with the window (PROP:SCROLL). Not valid in a REPORT.
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened. UNHIDE must be used to display it (PROP:HIDE). In a REPORT, specifies the control does not print unless UNHIDE is used to allow it to print
LINEWIDTH	Specifies the width of the BOX's border (PROP:LINEWIDTH).
LAYOUT	Specifies the control's left-to-right or right-to-left display orientation (PROP:LAYOUT)
EXTEND	Specifies special document formatting information. This information contains a valid string of attributes that are assigned to a designated REPORT control for a given document type. Valid only in a REPORT.

The **BOX** control places a rectangular box on the WINDOW, TOOLBAR, or REPORT at the position and size specified by its AT attribute. This control cannot receive input focus and does not generate events.

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
    !Unfilled, black border:
    BOX,AT(0,0,20,20)
    !Unfilled, black border, dimmed:
    BOX,AT(0,20,20,20),USE(?Box1),DISABLE
    !Unfilled, rounded, black border:
    BOX,AT(20,20,20,20),ROUND
    !Filled, black border:
    BOX,AT(40,40,20,20),FILL(COLOR:ACTIVEBORDER)
    !Unfilled, active border color border:
    BOX,AT(60,60,20,20),COLOR(COLOR:ACTIVEBORDER)
    !Scrolls with screen:
    BOX,AT(480,180,20,20),SCROLL
END
```

```
CustRpt REPORT,AT(1000,1000,6500,9000),THOUS
CustDetail DETAIL,AT(0,0,6500,1000)
    !Unfilled, black border
    BOX,AT(0,0,20,20),USE(?B1)
    !Unfilled, rounded, black border
    BOX,AT(20,20,20,20),ROUND
    !Filled, black border
    BOX,AT(40,40,20,20),FILL(COLOR:ACTIVEBORDER)
    !Unfilled, active border color border
    BOX,AT(60,60,20,20),COLOR(COLOR:ACTIVEBORDER)
    END
END
```

See Also:

PANEL

BUTTON (declare a pushbutton control)

BUTTON(*text*) ,**AT**() [,**CURSOR**()] [,**USE**()] [,**DISABLE**] [,**KEY**()] [,**MSG**()] [,**HLP**()] [,**SKIP**]
 [,**STD**()] [,**FONT**()] [,**ICON**()] [,**DEFAULT**] [,**IMM**][,**REQ**] [,**FULL**] [,**SCROLL**] [,**ALRT**()]
 [,**HIDE**] [,**DROPID**()] [,**TIP**()] [,**FLAT**] [,**REPEAT**()] [,**DELAY**()] [,| **LEFT**|] [,**TRN**] [,**LAYOUT**]
 | **RIGHT**|

BUTTON	Places a command button on the WINDOW or TOOLBAR.
<i>text</i>	A string constant containing the text to display on the button face, along with any ICON specified (PROP:Text). This may contain an ampersand (&) to indicate the "hot" letter (accelerator key) for the button.
AT	Specifies the initial size and location of the control (PROP:AT). If omitted, default values are set by the runtime library.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the control (PROP:CURSOR). If omitted, the WINDOW's CURSOR attribute is used, else the Windows default cursor is used.
USE	A field equate label to reference the control in executable code (PROP:USE).
DISABLE	Specifies the control appears dimmed when the WINDOW or APPLICATION is first opened (PROP:DISABLE).
KEY	Specifies an integer constant or keycode equate that immediately gives focus to and presses the button (PROP:KEY).
MSG	Specifies a string constant containing the text to display in the status bar when the control has focus (PROP:MSG).
HLP	Specifies a string constant containing the help system identifier for the control (PROP:HLP).
SKIP	Specifies the control does not receive input focus and may only be accessed with the mouse or accelerator key (PROP:SKIP).
STD	Specifies an integer constant or equate that identifies a "Windows standard action" the control executes (PROP:STD).
FONT	Specifies the display font for the control (PROP:FONT).
ICON	Specifies an image file or standard icon to display on the button face (PROP:ICON).
DEFAULT	Specifies the BUTTON is automatically pressed when the user presses the ENTER key (PROP:DEFAULT).

IMM	Specifies the control generates an event when the left mouse button is pressed, continuing as long as it is depressed (PROP:IMM). If omitted, an event is generated only when the left mouse button is pressed and released on the control.
REQ	Specifies that when the BUTTON is pressed, the runtime library automatically checks all ENTRY controls in the same WINDOW with the REQ attribute to ensure they contain data other than blanks or zeroes (PROP:REQ).
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter (PROP:FULL).
SCROLL	Specifies the control scrolls with the window (PROP:SCROLL).
ALRT	Specifies "hot" keys active for the control (PROP:ALRT).
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened (PROP:HIDE). UNHIDE must be used to display it.
DROPID	Specifies the control may serve as a drop target for drag-and-drop actions (PROP:DROPID).
TIP	Specifies the text that displays as "balloon help" when the mouse cursor pauses over the control (PROP:ToolTip).
FLAT	Specifies the button appears flat except when the mouse cursor passes over the control (PROP:FLAT). Requires the ICON attribute.
REPEAT	Specifies the rate at which EVENT:Accepted generates when the button with the IMM attribute is held down by the user (PROP:REPEAT). Requires the IMM attribute.
DELAY	Specifies the delay between the first and second generation of EVENT:Accepted for a button with the IMM attribute (PROP:DELAY). Requires the IMM attribute.
LEFT	Specifies that the icon appears to the left of the <i>text</i> (PROP:LEFT).
RIGHT	Specifies that the icon appears to the right of the <i>text</i> (PROP:RIGHT).
TRN	Specifies that the characters print or display transparently without obliterating the background over which the control is placed.
LAYOUT	Specifies the control's left-to-right or right-to-left display orientation (PROP:LAYOUT)

The **BUTTON** control places a pushbutton on the **WINDOW** or **TOOLBAR** (not valid in a **REPORT**) at the position and size specified by its **AT** attribute.

A **BUTTON** with the **IMM** attribute generates **EVENT:Accepted** as soon as the left mouse button is pressed on the control and continues to do so until it is released. This allows the **BUTTON** control's executable code to execute continuously until the mouse button is released. The rate and delay before continuous event generation can be set by the **REPEAT** and **DELAY** attributes.

A **BUTTON** without the **IMM** attribute generates **EVENT:Accepted** only when the left mouse button is pressed and then released on the control.

A **BUTTON** with the **REQ** attribute is a "required control fields check" button. **REQ** attributes of **ENTRY** or **TEXT** control fields are not checked until a **BUTTON** with the **REQ** attribute is pressed or the **INCOMPLETE** procedure is called. Focus is given to the first required control which is blank or zero.

A **BUTTON** with an **ICON** attribute displays the image on the button face in addition to its *text* parameter (which appears below the image, by default). The *text* parameter also serves for accelerator "hot" key definition.

Events Generated:

EVENT:Selected	The control has received input focus.
EVENT:Accepted	The control has been pressed by the user.
EVENT:PreAlertKey	The user pressed an ALRT attribute hot key.
EVENT:AlertKey	The user pressed an ALRT attribute hot key.
EVENT:Drop	A successful drag-and-drop to the control.

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
    BUTTON('1'),AT(0,0,20,20),USE(?B1)
    BUTTON('2'),AT(20,0,20,20),USE(?B2),KEY(F10Key)
    BUTTON('3'),AT(40,0,20,20),USE(?B3),MSG('Button 3')
    BUTTON('4'),AT(60,0,20,20),USE(?B4),HLP('Button4Help')
    BUTTON('5'),AT(80,0,20,20),USE(?B5),STD(STD:Cut)
    BUTTON('6'),AT(100,0,20,20),USE(?B6),FONT('Arial',12)
    BUTTON('7'),AT(120,0,20,20),USE(?B7),ICON(ICON:Question)
    BUTTON('8'),AT(140,0,20,20),USE(?B8),DEFAULT
    BUTTON('9'),AT(160,0,20,20),USE(?B9),IMM
    BUTTON('10'),AT(180,0,20,20),USE(?B10),CURSOR(CURSOR:Wait)
    BUTTON('11'),AT(200,0,20,20),USE(?B11),REQ
    BUTTON('12'),AT(220,0,20,20),USE(?B12),ALRT(F10Key)
    BUTTON('13'),AT(240,0,20,20),USE(?B13),SCROLL
END

CODE
OPEN(MDIChild)
ACCEPT
CASE ACCEPTED()
OF ?B1
!Perform some action
END
END
```

See Also: **CHECK**, **OPTION**, **RADIO**

CHECK (declare a checkbox control)

```

CHECK(text ,AT( ) [,CURSOR( )] [,USE( )] [,DISABLE] [,KEY( )] [,MSG( )] [,HLP( )] [,SKIP]
      [,FONT( )] [,ICON( )] [,FULL] [,SCROLL] [,ALRT( )] [,HIDE] [,DROPID( )] [,TIP( )]
      [,LEFT |] [,VALUE( )] [,TRN] [,COLOR( )] [,FLAT] [,LAYOUT] [,EXTEND( )]
      | RIGHT |

```

CHECK	Places a check box on the WINDOW, TOOLBAR, or REPORT.
<i>text</i>	A string constant containing the text to display next to the check box (PROP:Text). This may contain an ampersand (&) to indicate the "hot" letter for the check box.
AT	Specifies the initial size and location of the control (PROP:AT). If omitted, default values are selected by the runtime library.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the control (PROP:CURSOR). If omitted, the WINDOW's CURSOR attribute is used, else the Windows default cursor is used. Not valid in a REPORT.
USE	The label of a variable to receive the value of the check box (PROP:USE). Zero (0) indicates OFF (un-checked) or one (1) indicates ON (checked) unless the VALUE attribute specifies other values.
DISABLE	Specifies the control appears dimmed in the WINDOW or APPLICATION (PROP:DISABLE).
KEY	Specifies an integer constant or keycode equate that immediately gives focus to and toggles the box (PROP:KEY). Not valid in a REPORT.
MSG	Specifies a string constant containing the text to display in the status bar when the control has focus (PROP:MSG). Not valid in a REPORT.
HLP	Specifies a string constant containing the help system identifier for the control (PROP:HLP). Not valid in a REPORT.
SKIP	Specifies the control does not receive input focus and may only be accessed with the mouse or accelerator key (PROP:SKIP). Not valid in a REPORT.
FONT	Specifies the display font for the control (PROP:FONT).
ICON	Specifies an image file or standard icon to display on the button face of a "latching" pushbutton (PROP:ICON). Not valid in a REPORT.
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter (PROP:FULL). Not valid in a REPORT.

SCROLL	Specifies the control scrolls with the window (PROP:SCROLL). Not valid in a REPORT.
ALRT	Specifies "hot" keys active for the control (PROP:ALRT).
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened, or the control is not printed in the REPORT (PROP:HIDE). UNHIDE must be used to display or print it.
DROPID	Specifies the control may serve as a drop target for drag-and-drop actions (PROP:DROPID). Not valid in a REPORT.
TIP	Specifies the text that displays as "balloon help" when the mouse cursor pauses over the control (PROP:ToolTip). Not valid in a REPORT.
LEFT	Specifies that the <i>text</i> appears to the left of the check box (PROP:LEFT).
RIGHT	Specifies that the <i>text</i> appears to the right of the check box (PROP:RIGHT). This is the default position.
VALUE	Specifies the true and false values the USE variable receives when the box is checked by the user (PROP:Value).
TRN	Specifies the control transparently displays over the background (PROP:TRN).
COLOR	Specifies a background color for the control's text (PROP:COLOR).
FLAT	Specifies the button appears flat except when the mouse cursor passes over the control (PROP:FLAT). Requires the ICON attribute. Not valid in a REPORT.
LAYOUT	Specifies the control's left-to-right or right-to-left display orientation (PROP:LAYOUT)
EXTEND	Specifies special document formatting information. This information contains a valid string of attributes that are assigned to a designated REPORT control for a given document type. Valid only in a REPORT.

The **CHECK** control places a check box on the WINDOW, TOOLBAR, or REPORT at the position and size specified by its AT attribute.

A CHECK in a window with an ICON attribute appears as a "latched" button with the image displayed on the button face. When the button appears "up" the CHECK is off; when it appears "down" the CHECK is on.

By default, when the CHECK is off the USE variable receives a value of zero (0); and when the CHECK is on, the USE variable receives a value of one (1). The VALUE attribute and its runtime properties (PROP:TrueValue and PROP:FalseValue) can be used to change this default behavior and automatically set the USE variable to values other than the defaults.

Events Generated:

EVENT:Selected	The control has received input focus.
EVENT:Accepted	The control has been toggled by the user.
EVENT:PreAlertKey	The user pressed an ALRT attribute hot key.
EVENT:AlertKey	The user pressed an ALRT attribute hot key.
EVENT:Drop	A successful drag-and-drop to the control.

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
    CHECK('1'),AT(0,0,20,20),USE(C1)
    CHECK('2'),AT(0,20,20,20),USE(C2),VALUE('T','F')
END

CustRpt REPORT,AT(1000,1000,6500,9000),THOUS
CustDetail DETAIL,AT(0,0,6500,1000)
    CHECK('1'),AT(0,0,20,20),USE(C1)
    CHECK('2'),AT(20,80,20,20),USE(C2),LEFT
    CHECK('3'),AT(0,100,20,20),USE(C3),FONT('Arial',12)
END

CODE
OPEN(MDIChild)
ACCEPT
CASE ACCEPTED()
OF ?C1
    IF C1 = 1 THEN DO C1Routine.
OF ?C2
    IF C2 = 'T' THEN DO C2Routine.
END
END
```

See Also:

BUTTON

OPTION

RADIO

COMBO (declare an entry/list control)

COMBO(*picture*) ,**FROM**() ,**AT**() [,**CURSOR**()] [,**USE**()] [,**LAYOUT**()][,**DISABLE**] [,**KEY**()] [,**MSG**()] [,**HLP**()]
 [,**SKIP**][,**FONT**()][,**FORMAT**()][,**DROP**][,**COLUMN**][,**VCR**][,**FULL**][,**GRID**()][,**SCROLL**]
 [,**ALRT**()][,**HIDE**][,**READONLY**][,**REQ**][,**NOBAR**][**DROPID**()][,**TIP**()][,**TRN**][,**COLOR**()]
 [, **MARK**()] [, | **HSCROLL** |] [, | **LEFT** |] [, | **INS** |] [, | **UPR** |]
 [, **MASK** | **IMM** |] | **VSCROLL** | | **RIGHT** | | **OVR** | | **CAP** | |
 | **HVSCROLL** | | **CENTER** | |
 | **DECIMAL** | |

COMBO	Places a data entry field with an associated list of data items on the WINDOW or TOOLBAR.
<i>picture</i>	A display picture token that specifies the input format for the data entered into the control (PROP:Text).
FROM	Specifies the origin of the data displayed in the list (PROP:FROM).
AT	Specifies the initial size and location of the control (PROP:AT). If omitted, the runtime library chooses a value.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the control (PROP:CURSOR). If omitted, the WINDOW's CURSOR attribute is used, else the Windows default cursor is used.
USE	A field equate label to reference the control in executable code or the label of the variable that receives the value selected by the user (PROP:USE).
LAYOUT	Specifies the control's left-to-right or right-to-left display and entry orientation (PROP:LAYOUT)
DISABLE	Specifies the control appears dimmed when the WINDOW or APPLICATION is first opened (PROP:DISABLE).
KEY	Specifies an integer constant or keycode equate that immediately gives focus to the control (PROP:KEY).
MSG	Specifies a string constant containing the text to display in the status bar when the control has focus (PROP:MSG).
HLP	Specifies a string constant containing the help system identifier for the control (PROP:HLP).
SKIP	Specifies the control receives input focus to enter text only with the mouse or accelerator key and does not retain focus (PROP:SKIP).
FONT	Specifies the display font for the control (PROP:FONT).
FORMAT	Specifies the display format of the data (PROP:FORMAT).

DROP	Specifies a drop-down list box and the number of elements the drop-down portion contains (PROP:DROP).
COLUMN	Specifies a field-by-field highlight bar on multi-column list boxes (PROP:COLUMN).
VCR	Specifies a VCR-type control that appears left of any horizontal scroll bar (PROP:VCR).
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter (PROP:FULL).
GRID	Specifies the color of the grid lines between columns in the list (PROP:GRID).
SCROLL	Specifies the control scrolls with the window (PROP:SCROLL).
ALRT	Specifies "hot" keys active for the control (PROP:ALRT).
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened (PROP:HIDE). UNHIDE must be used to display it.
READONLY	Specifies the control does not allow data entry (PROP:READONLY).
NOBAR	Specifies the highlight bar is displayed only when the LIST has focus (PROP:NOBAR).
DROPID	Specifies the control may serve as a drop target for drag-and-drop actions (PROP:DROPID).
TIP	Specifies the text that displays as "balloon help" when the mouse cursor pauses over the control (PROP:ToolTip).
TRN	Specifies the control transparently displays over the background (PROP:TRN).
COLOR	Specifies background and selected colors for the control (PROP:COLOR).
REQ	Specifies the control may not be left blank or zero (PROP:REQ).
MARK	Specifies multiple item selection mode (PROP:MARK).
IMM	Specifies generation of an event whenever the user presses any key (PROP:IMM).
HSCROLL	Specifies that a horizontal scroll bar is automatically added to the list box when any portion of the data item lies horizontally outside the visible area (PROP:HSCROLL).
VSCROLL	Specifies that a vertical scroll bar is automatically added to the list box when any data items lie vertically outside the visible area (PROP:VSCROLL).
HVSCROLL	Specifies that both vertical and horizontal scroll bars are automatically added to the list box when any portion of the data items lies outside the visible area (PROP:HVSCROLL).
LEFT	Specifies that the data is left justified within the control (PROP:LEFT).
RIGHT	Specifies that the data is right justified within the control (PROP:RIGHT).

CENTER	Specifies that the data is centered within the control (PROP:CENTER).
DECIMAL	Specifies that the data is aligned on the decimal point within the control (PROP:DECIMAL).
INS / OVR	Specifies Insert or Overwrite entry mode (PROP:INS and PROP:OVR). This is valid only on windows with the MASK attribute.
UPR / CAP	Specifies all upper case or proper name capitalization (First Letter Of Each Word Capitalized) data entry (PROP:UPR and PROP:CAP).
MASK	Specifies pattern input editing mode of the ENTRY portion of the control (PROP:MASK).

The **COMBO** control places a data entry field with an associated list of data items on the WINDOW or TOOLBAR (not valid in a REPORT) at the position and size specified by its AT attribute (a combination of an ENTRY and LIST control). The user may type in data or select an item from the list. The entered data is not automatically validated against the entries in the list. The data entry portion of the COMBO acts as an "incremental locator" to the list--as the user types each character, the highlight bar is positioned to the closest matching entry.

A COMBO with the DROP attribute displays only the currently selected data item on screen until the control has focus and the user presses the down arrow key, or CLICKS ON the the icon to the right of the displayed data item. When either of these occurs, the selection list appears ("drops down") to allow the user to select an item.

A COMBO with the IMM attribute generates an EVENT:NewSelection every time the user moves the highlight bar to another selection, or presses any key (all keys are implicitly ALRTed). This allows an opportunity for the source code to re-fill the display QUEUE, or get the currently highlighted record to display other fields from the record. A COMBO with the VCR attribute has scroll control buttons like a **Video Cassette Recorder** to the left of the horizontal scroll bar (if there is one). These buttons allow the user to use the mouse to scroll through the list.

Events Generated:

EVENT:Selected	The control has received input focus.
EVENT:Accepted	The user has either selected an entry from the list or entered data directly into the control, and moved on to another control.
EVENT:Rejected	The user has entered an invalid value for the entry picture.
EVENT:NewSelection	The current selection in the list has changed (highlight bar has moved up or down) or the user pressed any key (only with the IMM attribute).
EVENT:PreAlertKey	The user pressed an ALRT attribute hot key.
EVENT:AlertKey	The user pressed an ALRT attribute hot key.
EVENT:Drop	A successful drag-and-drop to the control.

EVENT:ScrollUp	The user pressed the up arrow (only with the IMM attribute).
EVENT:ScrollDown	The user pressed the down arrow (only with the IMM attribute).
EVENT:PageUp	The user pressed PgUp (only with the IMM attribute).
EVENT:PageDown	The user pressed PgDn (only with the IMM attribute).
EVENT:ScrollTop	The user pressed Ctrl-PgUp (only with the IMM attribute).
EVENT:ScrollBottom	The user pressed Ctrl-PgDn (only with the IMM attribute).
EVENT:PreAlertKey	The user pressed a printable character (only with the IMM attribute) or an ALRT attribute hot key.
EVENT:AlertKey	The user pressed a printable character (only with the IMM attribute) or an ALRT attribute hot key.
EVENT:Locate	The user pressed the locator VCR button (only with the IMM attribute).
EVENT:ScrollDrag	The user moved the scroll bar's "thumb" and its new position is in PROP:VScrollPos (only with the IMM attribute).
EVENT:ScrollTrack	The user is moving the scroll bar's "thumb" and its new position is in PROP:VScrollPos (only with the IMM attribute).
EVENT:DroppingDown	The user pressed the down arrow button (only with the DROP attribute).
EVENT:DroppedDown	The list has dropped (only with the DROP attribute).
EVENT:ColumnResize	A column in the list has been resized.

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
    COMBO(@S8),AT(0,0,20,20),USE(C1),FROM(Que)
    COMBO(@S8),AT(20,0,20,20),USE(C2),FROM(Que),KEY(F10Key)
    COMBO(@S8),AT(40,0,20,20),USE(C3),FROM(Que),MSG('Button 3')
    COMBO(@S8),AT(60,0,20,20),USE(C4),FROM(Que),HLP('Check4Help')
    COMBO(@S8),AT(80,0,20,20),USE(C5),FROM(Q) |
    ,FORMAT('5C~List~15L~Box~'),COLUMN
    COMBO(@S8),AT(100,0,20,20),USE(C6),FROM(Que),FONT('Arial',12)
    COMBO(@S8),AT(120,0,20,20),USE(C7),FROM(Que),DROP(8)
    COMBO(@S8),AT(140,0,20,20),USE(C8),FROM(Que),HVSCROLL,VCR
    COMBO(@S8),AT(160,0,20,20),USE(C9),FROM(Que),IMM
    COMBO(@S8),AT(180,0,20,20),USE(C10),FROM(Que),CURSOR(CURSOR:Wait)
    COMBO(@S8),AT(200,0,20,20),USE(C11),FROM(Que),ALRT(F10Key)
    COMBO(@S8),AT(220,0,20,20),USE(C12),FROM(Que),LEFT
    COMBO(@S8),AT(240,0,20,20),USE(C13),FROM(Que),RIGHT
    COMBO(@S8),AT(260,0,20,20),USE(C14),FROM(Que),CENTER
    COMBO(@N8.2),AT(280,0,20,20),USE(C15),FROM(Que),DECIMAL
    COMBO(@S8),AT(300,0,20,20),USE(C16),FROM('Apples|Peaches|Pumpkin|Pie')
    COMBO(@S8),AT(320,0,20,20),USE(C17),FROM('TBA')
END
```

```
CODE
OPEN(MDICHild)
?C17{PROP:From} = 'Live|Long|And|Prosper'   !Runtime FROM attribute assignment
ACCEPT
CASE ACCEPTED()
OF ?C1
  LOOP X# = 1 to RECORDS(Que)                !Check for user's entry in Que
    GET(Que,X#)
    IF C1 = Que THEN BREAK.                  !Break loop if present
  END
  IF X# > RECORDS(Que)                        !Check for BREAK
    Que = C1                                 !and add the entry
    ADD(Que)
  END
END
END
END
```

See Also:

LIST

ENTRY

ELLIPSE (declare an ellipse control)

ELLIPSE ,AT() [,USE()] [,DISABLE] [,COLOR()] [,FILL()] [,FULL] [,SCROLL] [,HIDE] [,LAYOUT()]
[,LINEWIDTH] [,EXTEND()]

ELLIPSE	Places a "circular" figure on the WINDOW, TOOLBAR, or REPORT.
AT	Specifies the initial size and location of the control (PROP:AT). If omitted, default values are selected by the runtime library.
USE	Specifies a field equate label to reference the control in executable code (PROP:USE).
DISABLE	Specifies the control appears dimmed when the WINDOW or APPLICATION is first opened (PROP:DISABLE).
COLOR	Specifies the color for the border of the ellipse (PROP:COLOR). If omitted, the ellipse has no border.
FILL	Specifies the fill color for the control (PROP:FILL). If omitted, the ellipse is not filled with color.
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter (PROP:FULL). Not valid in a REPORT.
SCROLL	Specifies the control scrolls with the window (PROP:SCROLL). Not valid in a REPORT.
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened (PROP:HIDE). UNHIDE must be used to display it.
LAYOUT	Specifies the control's left-to-right or right-to-left display orientation (PROP:LAYOUT)
LINEWIDTH	Specifies the width of the ELLIPSE's border (PROP:LINEWIDTH).
EXTEND	Specifies special document formatting information. This information contains a valid string of attributes that are assigned to a designated REPORT control for a given document type. Valid only in a REPORT.

The **ELLIPSE** control places a "circular" figure on the WINDOW, TOOLBAR, or REPORT at the position and size specified by its AT attribute. The ellipse is drawn inside a "bounding box" defined by the *x*, *y*, *width*, and *height* parameters of its AT attribute. The *x* and *y* parameters specify the starting point, and the *width* and *height* parameters specify the horizontal and vertical size of the "bounding box." This control cannot receive input focus and does not generate events.

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
    !Filled, full screen, black border:
    ELLIPSE,FILL(COLOR:MENU),FULL
    !Unfilled, black border:
    ELLIPSE,AT(0,0,20,20)
    !Dimmed:
    ELLIPSE,AT(0,20,20,20),USE(?Box1),DISABLE
    !Unfilled, rounded, black border:
    ELLIPSE,AT(20,20,20,20),ROUND
    !Filled, black border:
    ELLIPSE,AT(40,40,20,20),FILL(COLOR:ACTIVEBORDER)
    !Unfilled, active border color border:
    ELLIPSE,AT(60,60,20,20),COLOR(COLOR:ACTIVEBORDER)
    !Scrolls with screen:
    ELLIPSE,AT(480,180,20,20),SCROLL
END
```

```
CustRpt REPORT,AT(1000,1000,6500,9000),THOUS
CustDetail DETAIL,AT(0,0,6500,1000)
    !Unfilled, black border:
    ELLIPSE,AT(0,0,20,20)
    !Unfilled, black border, dimmed:
    ELLIPSE,AT(0,20,20,20),USE(?Ellipse1),DISABLE
    !Unfilled, rounded, black border:
    ELLIPSE,AT(20,20,20,20),ROUND
    !Filled, black border:
    ELLIPSE,AT(40,40,20,20),FILL(COLOR:ACTIVEBORDER)
    !Unfilled, active border color border
    ELLIPSE,AT(60,60,20,20),COLOR(COLOR:ACTIVEBORDER)
    END
END
```

ENTRY (declare a data entry control)

ENTRY(*picture*) ,**AT**() [, **CURSOR**()] [, **USE**()] [, **LAYOUT**()] [, **DISABLE**] [, **KEY**()] [, **MSG**()] [, **HLP**()] [, **SKIP**] [, **FONT**()] [, **IMM**] [, **PASSWORD**] [, **REQ**] [, **FULL**] [, **SCROLL**] [, **ALRT**()] [, **HIDE**] [, **TIP**()] [, **TRN**] [, **READONLY**] [**DROPID**()] [, | **INS** |] [, | **CAP** |] [, | **LEFT** |] [, **COLOR**()] [, **MASK**] | **OVR** | | **UPR** | | **RIGHT** | | **CENTER** | | **DECIMAL** |

ENTRY	Places a data entry field on the WINDOW or TOOLBAR.
<i>picture</i>	A display picture token that specifies the input format for the data entered into the control (PROP:Text).
AT	Specifies the initial size and location of the control (PROP:AT). If omitted, default values are selected by the runtime library.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the control (PROP:CURSOR). If omitted, the WINDOW's CURSOR attribute is used, else the Windows default cursor is used.
USE	The label of the variable that receives the value entered into the control by the user (PROP:USE).
LAYOUT	Specifies the control's left-to-right or right-to-left display and entry orientation (PROP:LAYOUT)
DISABLE	Specifies the control appears dimmed when the WINDOW or APPLICATION opens (PROP:DISABLE).
KEY	Specifies an integer constant or keycode equate that immediately gives focus to the control (PROP:KEY).
MSG	Specifies a string constant containing the text to display in the status bar when the control has focus (PROP:MSG).
HLP	Specifies a string constant containing the help system identifier for the control (PROP:HLP).
SKIP	Specifies the control receives input focus to enter text only with the mouse or accelerator key and does not retain focus (PROP:SKIP).
FONT	Specifies the display font for the control (PROP:FONT).
IMM	Specifies immediate event generation whenever the user presses any key (PROP:IMM).

PASSWORD	Specifies non-display of the data entered (password mode) (PROP:PASSWORD).
REQ	Specifies the control may not be left blank or zero (PROP:REQ).
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter (PROP:FULL).
SCROLL	Specifies the control scrolls with the window (PROP:SCROLL).
ALRT	Specifies "hot" keys active for the control (PROP:ALRT).
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened (PROP:HIDE). UNHIDE must be used to display it.
TIP	Specifies the text that displays as "balloon help" when the mouse cursor pauses over the control (PROP:ToolTip).
TRN	Specifies the control transparently displays over the background (PROP:TRN).
READONLY	Specifies the control does not allow data entry (PROP:READONLY).
DROPID	Specifies the control may serve as a drop target for drag-and-drop actions (PROP:DROPID).
INS / OVR	Specifies Insert or Overwrite entry mode (PROP:INS and PROP:OVR). This is valid only on windows with the MASK attribute.
UPR / CAP	Specifies all upper case or proper name capitalization (First Letter Of Each Word Capitalized) data entry (PROP:UPR and PROP:CAP).
LEFT	Specifies that the data entered is left justified within the area specified by the AT attribute (PROP:LEFT).
RIGHT	Specifies that the data entered is right justified within the area specified by the AT attribute (PROP:RIGHT).
CENTER	Specifies that the data entered is centered within the area specified by the AT attribute (PROP:CENTER).
DECIMAL	Specifies that the data entered is aligned on the decimal point within the area specified by the AT attribute (PROP:DECIMAL).
COLOR	Specifies background and selected colors for the control (PROP:COLOR).
MASK	Specifies pattern input editing mode of the ENTRY control (PROP:MASK).

The **ENTRY** control places a data entry field on the WINDOW or TOOLBAR (not valid in a REPORT) at the position and size specified by its AT attribute. Data entered is formatted according to the *picture*, and the variable specified in the USE attribute receives the data entered when the user has completed data entry and moves on to another control. Data entry scrolls horizontally to allow the user to enter data to the full length of the variable. Therefore, the right and left arrow keys move within the data in the ENTRY control.

Standard Windows behavior (Cut, Copy, and Paste) are automatically available using CTRL+X, CTRL+C, and CTRL+V while the ENTRY control has focus. Undo is also implemented using CTRL+Z (before the user leaves the control).

An ENTRY control with the PASSWORD attribute displays asterisks when the user enters data (and Cut and Copy are disabled). This is useful for password-type variables. An ENTRY control with the SKIP attribute is used for seldom-used data entry. Display-only data should be declared with the READONLY attribute.

The LAYOUT attribute is used with ENTRY controls to change the order that text is entered (Left-to-Right or Right-to-Left). Mixed text should always be avoided and may not display correctly (e.g. Arabic text mixed with English text).

The MASK attribute specifies pattern input editing mode of the control. This means that, as the user types in data, each character is automatically validated against the control's picture for proper input (numbers only in numeric pictures, etc.). This forces the user to enter data in the format specified by the control's display picture. If omitted, Windows free-input is allowed in the control. This is Windows' default data entry mode. Free-input means the user's data is formatted to the control's picture only after entry (on EVENT:Accepted). This allows users to enter data as they choose and it is automatically formatted to the control's picture after entry. If the user types in data in a format different from the control's picture, the libraries attempt to determine the format the user used, and convert the data to the control's display picture. For example, if the user types "January 1, 1995" into a control with a display picture of @D1, the runtime library formats the user's input to "1/1/95." This action occurs only after the user completes data entry and moves to another control. If the runtime library cannot determine what format the user used, it will not update the USE variable and will simply generate EVENT:Rejected.

Events Generated:

EVENT:Selected	The control has received input focus.
EVENT:Accepted	The user has completed data entry in the control.
EVENT:Rejected	The user has entered an invalid value for the entry picture.
EVENT:PreAlertKey	The user pressed an ALRT attribute hot key.
EVENT:AlertKey	The user pressed an ALRT attribute hot key.
EVENT:Drop	A successful drag-and-drop to the control.
EVENT:NewSelection	The user entered a character (with IMM attribute only).

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
    ENTRY(@S8),AT(0,0,20,20),USE(E1)
    ENTRY(@S8),AT(20,0,20,20),USE(E2),KEY(F10Key)
    ENTRY(@S8),AT(40,0,20,20),USE(E3),MSG('Button 3')
    ENTRY(@S8),AT(60,0,20,20),USE(E4),HLP('Entry4Help')
    ENTRY(@S8),AT(80,0,20,20),USE(E5),DISABLE
    ENTRY(@S8),AT(100,0,20,20),USE(E6),FONT('Arial',12)
    ENTRY(@S8),AT(120,0,20,20),USE(E7),REQ,INS,CAP
    ENTRY(@S8),AT(140,0,20,20),USE(E8),SCROLL,OVR,UPR
    ENTRY(@S8),AT(180,0,20,20),USE(E9),CURSOR(CURSOR:Wait),IMM
    ENTRY(@S8),AT(200,0,20,20),USE(E10),ALRT(F10Key)
    ENTRY(@N8.2),AT(280,0,20,20),USE(E11),DECIMAL(10)
END
```

See Also:

TEXT

PROMPT

GROUP (declare a group of controls)

```

GROUP(text) ,AT() [,CURSOR()][,USE()] [,LAYOUT( )] [,DISABLE][,KEY()][,MSG()][,HLP()][,FONT()][,TIP()]
    controls
    [,BOXED][,FULL][,SCROLL][,HIDE][,ALRT()][,SKIP][,DROPID()][,COLOR()][,BEVEL()]
END

```

GROUP	Declares a group of controls that may be referenced as one entity.
<i>text</i>	A string constant containing the prompt for the group of controls (PROP:Text). This may contain an ampersand (&) to indicate the "hot" letter for the prompt. The <i>text</i> is displayed on screen only if the BOXED attribute is also present.
AT	Specifies the initial size and location of the control (PROP:AT). If omitted, default values are selected by the runtime library.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the control, or any control within the GROUP (PROP:CURSOR). If omitted, the window's CURSOR attribute is used, else the Windows default cursor is used. Not valid in a REPORT.
USE	A field equate label to reference the control in executable code (PROP:USE).
LAYOUT	Specifies the control's left-to-right or right-to-left display orientation (PROP:LAYOUT)
DISABLE	Specifies the GROUP control and the controls in the GROUP appear dimmed when the WINDOW or APPLICATION is first opened (PROP:DISABLE). Not valid in a REPORT.
KEY	Specifies an integer constant or keycode equate that immediately gives focus to the first control in the GROUP (PROP:KEY). Not valid in a REPORT.
MSG	Specifies a string constant containing the default text to display in the status bar when any control in the GROUP has focus (PROP:MSG). Not valid in a REPORT.
HLP	Specifies a string constant containing the default help system identifier for any control in the GROUP (PROP:HLP). Not valid in a REPORT.
FONT	Specifies the display font for the control and the default for all the controls in the GROUP (PROP:FONT).
TIP	Specifies the text that displays as "balloon help" when the mouse cursor pauses over the control (PROP:ToolTip). Not valid in a REPORT.
BOXED	Specifies a single-track border around the group of controls with the text at the top of the border (PROP:BOXED).

FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter (PROP:FULL).
SCROLL	Specifies the GROUP control and the controls in the GROUP scroll with the window (PROP:SCROLL).
HIDE	Specifies the GROUP control and the controls in the GROUP do not appear when the WINDOW or APPLICATION is first opened (PROP:HIDE). UNHIDE must be used to display them.
ALRT	Specifies "hot" keys active for the controls in the GROUP (PROP:ALRT).
SKIP	Specifies the controls in the GROUP do not receive input focus and may only be accessed with the mouse or accelerator key (PROP:SKIP). Not valid in a REPORT.
DROPID	Specifies the control may serve as a drop target for drag-and-drop actions (PROP:DROPID). Not valid in a REPORT.
COLOR	Specifies default background and selected colors for the controls in the GROUP (PROP:COLOR).
BEVEL	Specifies custom 3-D effect borders (PROP:BEVEL). Not valid in a REPORT.
<i>controls</i>	Control declarations that may be referenced as the GROUP.

The **GROUP** control declares a group of controls to reference as one entity. GROUP allows the user to use the cursor keys instead of the TAB key to move between the *controls* in the GROUP, and provides default MSG and HLP attributes for all controls in the GROUP. This control cannot receive input focus.

Events Generated:

EVENT:Drop A successful drag-and-drop to the control.

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
  GROUP('Group 1'),USE(?G1),KEY(F10Key)
    ENTRY(@S8),AT(0,0,20,20),USE(?E1)
    ENTRY(@S8),AT(20,0,20,20),USE(?E2)
  END
  GROUP('Group 2'),USE(?G2),MSG('Group 2'),CURSOR(CURSOR:Wait)
    ENTRY(@S8),AT(40,0,20,20),USE(?E3)
    ENTRY(@S8),AT(60,0,20,20),USE(?E4)
  END
  GROUP('Group 3'),USE(?G3),AT(80,0,20,20),BOXED
    ENTRY(@S8),AT(80,0,20,20),USE(?E5)
    ENTRY(@S8),AT(100,0,20,20),USE(?E6)
  END
END
```

```
CustRpt  REPORT,AT(1000,1000,6500,9000),THOUS
CustDetail  DETAIL,AT(0,0,6500,1000)
          GROUP('Group 1'),USE(!G1),AT(80,0,20,20),BOXED
            STRING(@S8),AT(80,0,20,20),USE(E5)
            STRING(@S8),AT(100,0,20,20),USE(E6)
          END
          GROUP('Group 2'),USE(?G2),FONT('Arial',12)
            STRING(@S8),AT(120,0,20,20),USE(E7)
            STRING(@S8),AT(140,0,20,20),USE(E8)
          END
        END
      END
```

See Also:

PANEL

IMAGE (declare a graphic image control)

```

IMAGE(file) ,AT ( ) [,USE ( )] [,DISABLE] [,FULL] [,SCROLL] [,HIDE] [,LAYOUT]
                                     [, TILED | ] [, HSCROLL | ] [,EXTEND ( )]
                                     | CENTERED | | VSCROLL |
                                     | HVSCROLL |

```

IMAGE	Places a graphic image on the WINDOW, TOOLBAR, or REPORT.
<i>file</i>	A string constant containing the name of the file to display (PROP:Text). The named file is automatically linked into the .EXE as a resource.
AT	Specifies the initial size and location of the control (PROP:AT). If omitted, default values are selected by the runtime library.
USE	A field equate label to reference the control in executable code (PROP:USE).
DISABLE	Specifies the control appears dimmed when the WINDOW or APPLICATION opens (PROP:DISABLE).
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter (PROP:FULL). Not valid in a REPORT.
SCROLL	Specifies the control scrolls with the window (PROP:SCROLL). Not valid in a REPORT.
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION opens (PROP:HIDE). UNHIDE must be used to display it.
TILED	Specifies the image displays at its default size and is tiled to fill the entire area of the IMAGE (PROP:TILED).
CENTERED	Specifies the image displays at its default size and is centered in the area of the IMAGE (PROP:CENTERED).
HSCROLL	Specifies a horizontal scroll bar is automatically added to the IMAGE control when the graphic image is wider than the area specified for display (PROP:HSCROLL). Not valid in a REPORT.
VSCROLL	Specifies a vertical scroll bar is automatically added to the IMAGE control when the graphic image is taller than the area specified for display (PROP:VSCROLL). Not valid in a REPORT.
HVSCROLL	Specifies both vertical and horizontal scroll bars are automatically added to the IMAGE control when the graphic image is larger than the display area (PROP:HVSCROLL). Not valid in a REPORT.
LAYOUT	Specifies the control's left-to-right or right-to-left display orientation (PROP:LAYOUT)

EXTEND Specifies special document formatting information. This information contains a valid string of attributes that are assigned to a designated REPORT control for a given document type. Valid only in a REPORT.

The **IMAGE** control places a graphic image on the WINDOW (or TOOLBAR) at the position specified by its AT attribute. The image is stretched to fill the area specified by the AT attribute unless the TILED or CENTERED attribute is present. The displayed *file* may be a bitmap (.BMP), PaintBrush (.PCX), Graphic Interchange Format (.GIF), JPEG (.JPG), or Windows metafile (.WMF). The *file* may be an icon (.ICO) in an IMAGE on a WINDOW but not on a REPORT, because Windows does not support printing icons. The type of *file* is determined by its extension.

This control cannot receive input focus and does not generate events.

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
    IMAGE('PIC.BMP'),AT(0,0,20,20),USE(?I1)
    IMAGE('PIC.WMF'),AT(40,0,20,20),USE(?I3),SCROLL
END
```

```
CustRpt REPORT,AT(1000,1000,6500,9000),THOUS
CustDetail DETAIL,AT(0,0,6500,1000)
    IMAGE('PIC.BMP'),AT(0,0,20,20),USE(?I1)
    IMAGE('PIC.WMF'),AT(40,0,20,20),USE(?I2)
    IMAGE('PIC.JPG'),AT(60,0,20,20),USE(?I3)
END
END
```

See Also:

PALETTE

ITEM (declare a menu item)

```
ITEM(text) [,AT( )] [,USE( )] [,KEY( )] [,MSG( )] [,HLP( )] [,STD( )] [,CHECK] [,DISABLE] [,COLOR] [,LEFT( )]
      [,SEPARATOR][,ICON( )] [,FONT( )] [,FIRST
      | LAST
      | ]
```

ITEM	Declares a menu choice within a MENUBAR or MENU structure.
<i>text</i>	A string constant containing the display text for the menu item (PROP:Text).
AT	Specifies the initial size and location of the menu item (PROP:AT). If omitted, default values are selected by the runtime library.
USE	A field equate label to reference the menu item in executable code, or the variable used with CHECK (PROP:USE).
KEY	Specifies an integer constant or keycode equate that immediately executes the menu item (PROP:KEY).
MSG	Specifies a string constant containing the text to display in the status bar when the menu item is highlighted (PROP:MSG).
HLP	Specifies a string constant containing the help system identifier for the menu item (PROP:HLP).
STD	Specifies an integer constant or equate that identifies a "Windows standard action" the menu item executes (PROP:STD).
CHECK	Specifies an on/off ITEM (PROP:CHECK).
DISABLE	Specifies the menu item appears dimmed when the WINDOW or APPLICATION is first opened (PROP:DISABLE).
COLOR	Specifies a background color for the control (PROP:COLOR).
LEFT	Specifies the offset in dialog units from the left edge of the menu structure.
SEPARATOR	Specifies the ITEM displays a solid horizontal line across the menu box at runtime to delimit groups of menu selections. The USE attribute may be specified with SEPARATOR.
ICON	Specifies an image file or standard icon to display on the menu item (PROP:ICON).
FONT	Specifies the display font for the control (PROP:FONT).
FIRST	Specifies the ITEM appears at the top of the menu when menus are merged (PROP:FIRST).
LAST	Specifies the ITEM appears at the bottom of the menu when menus are merged (PROP:LAST).

ITEM declares a menu choice within a MENUBAR or MENU structure. The *text* string may contain an ampersand (&) which designates the following character as an accelerator "hot" key which is automatically underlined. If the ITEM is on the menu bar, pressing the Alt key together with the accelerator key highlights and executes the ITEM. If the ITEM is in a MENU, pressing the accelerator key, alone, when the menu is displayed, highlights and executes the ITEM. If there is no ampersand in the *text*, the first non-blank character in the *text* string is the accelerator key for the ITEM, which will not be underlined. To include an ampersand as part of the *text*, place two ampersands together (&&) in the *text* string and only one will display. The KEY attribute designates a separate "hot" key for the item. This may be any valid Clarion keycode to immediately execute the ITEM's action.

A cursor bar highlights individual ITEMS within the MENU structure. Each ITEM is usually associated with some code to be executed upon selection of that ITEM, unless the STD attribute is present. The STD attribute specifies a standard Windows action the menu item performs, such as Tile or Cascade the windows. The SEPARATOR attribute creates an ITEM which serves only to delimit groups of menus selections so it should not have a *text* parameter. The USE attribute may be used with a SEPARATOR. It creates a solid horizontal line across the menu box. An ITEM that is not within a MENU structure is placed on the menu bar. This creates a menu bar selection which has no related drop-down menu. The normal convention to indicate this to the user is to terminate the *text* displayed for the item with an exclamation point (!). For example, the *text* for the ITEM might contain 'Exit!' to alert the user to the executable nature of the menu choice.

Events Generated:

EVENT:Accepted The control has been pressed by the user.

Example:

```

MainWin APPLICATION('My Application'),SYSTEM,MAX,ICON('MyIcon.ICO'),STATUS,HVSCROLL,RES
  MENUBAR
    ITEM('E&xit!'),USE(?MainExit),FIRST
    MENU('File'),USE(?FileMenu),FIRST
      ITEM('Open...'),USE(?OpenFile),HLP('OpenFileHelp'),FIRST
      ITEM('Close'),USE(?CloseFile),HLP('CloseFileHelp'),DISABLE
      ITEM('Auto Increment'),USE(ToggleVar),CHECK
    END
    MENU('Edit'),USE(?EditMenu),KEY(CtrlE),HLP('EditMenuHelp')
      ITEM('Undo'),USE(?UndoText),KEY(CtrlZ),STD(STD:Undo),DISABLE
      ITEM,SEPARATOR
      ITEM('Cu&t'),USE(?CutText),KEY(CtrlX),STD(STD:Cut),DISABLE
      ITEM('Copy'),USE(?CopyText),KEY(CtrlC),STD(STD:Copy),DISABLE
      ITEM('Paste'),USE(?PasteText),KEY(CtrlV),STD(STD:Paste),DISABLE
    END
    MENU('Window'),STD(STD:WindowList),MSG('Arrange or Select Window'),LAST
      ITEM('Tile'),STD(STD:TileWindow)
      ITEM('Cascade'),STD(STD:CascadeWindow)
      ITEM('Arrange Icons'),STD(STD:ArrangeIcons)
      ITEM,SEPARATOR,USE(?FileSeparator1)
    END
    MENU('Help'),USE(?HelpMenu),LAST,RIGHT
      ITEM('Contents'),USE(?HelpContents),STD(STD:HelpIndex)
      ITEM('Search for Help On...'),USE(?HelpSearch),STD(STD:HelpSearch)
      ITEM('How to Use Help'),USE(?HelpOnHelp),STD(STD:HelpOnHelp)
      ITEM('About MyApp...'),USE(?HelpAbout),MSG('Copyright Info'),LAST
    END
  END !Menubar
END !Application

```

LINE (declare a line control)

LINE .AT() [,USE()] [,DISABLE] [,COLOR()] [,FULL] [,SCROLL] [,HIDE] [,LINEWIDTH()]
[,LAYOUT()] [,EXTEND()]

LINE	Places a straight line on the WINDOW, TOOLBAR, or REPORT.
AT	Specifies the initial size and location of the control (PROP:AT). If omitted, default values are selected by the runtime library.
USE	A field equate label to reference the control in executable code (PROP:USE).
DISABLE	Specifies the control appears dimmed when the WINDOW or APPLICATION opens (PROP:DISABLE).
COLOR	Specifies the color for the line (PROP:COLOR). If omitted, the color is black.
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter (PROP:FULL). Not valid in a REPORT.
SCROLL	Specifies the control scrolls with the window (PROP:SCROLL). Not valid in a REPORT.
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened (PROP:HIDE). UNHIDE must be used to display it.
LINEWIDTH	Specifies the thickness of the LINE (PROP:LINEWIDTH).
LAYOUT	Specifies the control's left-to-right or right-to-left display orientation (PROP:LAYOUT)
EXTEND	Specifies special document formatting information. This information contains a valid string of attributes that are assigned to a designated REPORT control for a given document type. Valid only in a REPORT.

The **LINE** control places a straight line on the WINDOW, TOOLBAR, or REPORT at the position and size specified by its AT attribute. The *x* and *y* parameters of the AT attribute specify the starting point of the line. The *width* and *height* parameters of the AT attribute specify the horizontal and vertical distance to the end point of the line. If these are both positive numbers, the line slopes to the right and down from its starting point. If the *width* is negative, the line slopes left; if the *height* is negative, the line slopes left. If either the *width* or *height* is zero, the line is horizontal or vertical. This control cannot receive input focus and does not generate events.

<u>Width</u>	<u>Height</u>	<u>Result</u>
positive	positive	right and down from start point
negative	positive	left and down from start point
positive	negative	right and up from start point
negative	negative	left and up from start point
zero	positive	vertical, down from start point
zero	negative	vertical, up from start point
positive	zero	horizontal, right from start point
negative	zero	horizontal, left from start point

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
    LINE,AT(60,60,20,20),COLOR(COLOR:ACTIVEBORDER) !Border color
    LINE,AT(480,180,20,20),SCROLL !Scrolls with screen
END

CustRpt REPORT,AT(1000,1000,6500,9000),THOUS
CustDetail DETAIL,AT(0,0,6500,1000)
    LINE,AT(60,60,20,20),COLOR(COLOR:ACTIVEBORDER) !Border color
    LINE,AT(480,180,20,20),USE(?L2)
    END
END
```

LIST (declare a window list control)

```

LIST ,FROM( ) ,AT( ) [,CURSOR( )] [,USE( )] [,DISABLE] [,KEY( )] [,MSG( )] [,HLP( )] [,SKIP]
    [,FONT( )] [,FORMAT( )] [,DROP] [,COLUMN] [,VCR] [,FULL] [,SCROLL] [,NOBAR]
    [,ALRT( )] [,HIDE] [,DRAGID( )] [,DROPID( )] [,TIP( )] [,GRID( )] [,TRN] [,COLOR( )] [,LAYOUT( )]
    [,
        | MARK( )      | ] [,
        | IMM          |
        | HSCROLL      | ] [,
        | VSCROLL      |
        | HVSCROLL     |
        | LEFT         | ] [,
        | RIGHT        |
        | CENTER       |
        | DECIMAL      |
    ]

```

LIST	Places a scrolling list of data items on the WINDOW, TOOLBAR, or REPORT.
FROM	Specifies the origin of the data displayed in the list (PROP:FROM).
AT	Specifies the initial size and location of the control (PROP:AT). If omitted, the runtime library chooses a value.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the control (PROP:CURSOR). If omitted, the WINDOW's CURSOR attribute is used, else the Windows default cursor is used. Not valid in a REPORT.
USE	A field equate label to reference the control in executable code, or the label of the variable that receives the value selected by the user (PROP:USE).
DISABLE	Specifies the control appears dimmed when the WINDOW or APPLICATION opens (PROP:DISABLE).
KEY	Specifies an integer constant or keycode equate that immediately gives focus to the control (PROP:KEY). Not valid in a REPORT.
MSG	Specifies a string constant containing the text to display in the status bar when the control has focus (PROP:MSG). Not valid in a REPORT.
HLP	Specifies a string constant containing the help system identifier for the control (PROP:HLP). Not valid in a REPORT.
SKIP	Specifies the control does not receive input focus and may only be accessed with the mouse or accelerator key (PROP:SKIP). Not valid in a REPORT.
FONT	Specifies the display font for the control (PROP:FONT).
FORMAT	Specifies the display format of the data in the list (PROP:FORMAT). This can include icons, colors, and tree controls.
DROP	Specifies a drop-down list box and the number of elements the drop-down portion contains (PROP:DROP). Not valid in a REPORT.
COLUMN	Specifies cell-by-cell highlighting on multi-column lists (PROP:COLUMN). Not valid in a REPORT.

VCR	Specifies a VCR-type control to the left of any horizontal scroll bar (PROP:VCR). Not valid in a REPORT.
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter (PROP:FULL). Not valid in a REPORT.
SCROLL	Specifies the control scrolls with the window (PROP:SCROLL). Not valid in a REPORT.
NOBAR	Specifies the highlight bar is displayed only when the LIST has focus (PROP:NOBAR). Not valid in a REPORT.
ALRT	Specifies "hot" keys active for the control (PROP:ALRT). Not valid in a REPORT.
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION opens (PROP:HIDE). UNHIDE must be used to display it.
DRAGID	Specifies the control may serve as a drag host for drag-and-drop actions (PROP:DRAGID). Not valid in a REPORT.
DROPID	Specifies the control may serve as a drop target for drag-and-drop actions (PROP:DROPID). Not valid in a REPORT.
TIP	Specifies the text that displays as "balloon help" when the mouse cursor pauses over the control (PROP:ToolTip). Not valid in a REPORT.
GRID	Specifies the color of the grid lines between columns in the list (PROP:GRID).
TRN	Specifies the control transparently displays over the background (PROP:TRN).
COLOR	Specifies background and selected colors for the control (PROP:COLOR).
LAYOUT	Specifies the control's left-to-right or right-to-left display orientation (PROP:LAYOUT)
MARK	Specifies multiple items selection mode (PROP:MARK). Not valid in a REPORT.
IMM	Specifies generation of an event whenever the user presses any key (PROP:IMM). Not valid in a REPORT.
HSCROLL	Specifies that a horizontal scroll bar is automatically added to the list box when any portion of the data item lies horizontally outside the visible area (PROP:HSCROLL). Not valid in a REPORT.
VSCROLL	Specifies that a vertical scroll bar is automatically added to the list box when any data items lie vertically outside the visible area (PROP:VSCROLL). Not valid in a REPORT.
HVSCROLL	Specifies that both vertical and horizontal scroll bars are automatically added to the list box when any portion of the data items lies outside the visible area (PROP:HVSCROLL). Not valid in a REPORT.
LEFT	Specifies that the data is left justified within the LIST (PROP:LEFT).
RIGHT	Specifies that the data is right justified within the LIST (PROP:RIGHT).

- CENTER** Specifies that the data is centered within the LIST (PROP:CENTER).
- DECIMAL** Specifies that the data is aligned on the decimal point within the LIST (PROP:DECIMAL).

The **LIST** control places a scrolling list of data items on the WINDOW, TOOLBAR, or REPORT at the position and size specified by its AT attribute. The data items displayed in the LIST come from a QUEUE or STRING specified by the FROM attribute and are formatted by the parameters specified in the FORMAT attribute (which can include colors, icons, and tree control parameters).

The CHOICE procedure returns the QUEUE entry number (the value returned by POINTER(queue)) of the selected item when the EVENT:Accepted event has been generated by the LIST. The data displayed in the LIST is automatically refreshed every time through the ACCEPT loop, whether the AUTO attribute is present or not.

A LIST with the DROP attribute displays only the currently selected data item on screen until the control has focus and the user presses the down arrow key, or CLICKS ON the the icon to the right of the displayed data item. When either of these occurs, the selection list appears ("drops down") to allow the user to select an item.

A LIST with the IMM attribute generates an event every time the user moves the highlight bar to another selection, or presses any key (all keys are implicitly ALRTed). This allows an opportunity for the source code to re-fill the display QUEUE, or get the currently highlighted record to display other fields from the record. If VSCROLL is also present, the vertical scroll bar is always displayed and when the end-user CLICKS on the scroll bar, events are generated but the list does not move (executable code should perform this action). You can interrogate the PROP:VscrollPos property to determine the scroll thumb's position from 0 (top) to 255 (bottom).

A LIST with the VCR attribute has scroll control buttons like a **V**ideo **C**assette **R**ecorder to the left of the horizontal scroll bar (if there is one). These buttons allow the user to use the mouse to scroll through the list.

A LIST with the DRAGID attribute can serve as a drag-and-drop host, providing information to be moved or copied to another control. A LIST with the DROPID attribute can serve as a drag-and-drop target, receiving information from another control. These attributes work together to specify drag-and-drop "signatures" that define a valid target for the operation. The DRAGID() and DROPID() procedures, along with the SETDROPID procedure, are used to perform the data exchange.

REPORT Usage

LIST is valid only in a DETAIL structure. Its purpose is to allow the report format to duplicate the screen appearance of the LIST's FORMAT setting. When the first instance of the DETAIL structure containing the LIST prints, any headers in the FORMAT attribute print along with the current FROM attribute entry. When the last DETAIL structure containing the LIST prints, the LIST footers print along with the current FROM attribute entry.

Events Generated:

EVENT:Selected	The control has received input focus.
EVENT:Accepted	The user has selected an entry from the control.
EVENT:NewSelection	The current selection in the list has changed (the highlight bar has moved up or down).
EVENT:ScrollUp	The user pressed the up arrow (only with the IMM attribute).
EVENT:ScrollDown	The user pressed the down arrow (only with the IMM attribute).
EVENT:PageUp	The user pressed PGUP (only with the IMM attribute).
EVENT:PageDown	The user pressed PGDN (only with the IMM attribute).
EVENT:ScrollTop	The user pressed CTRL+PGUP (only with IMM attribute).
EVENT:ScrollBottom	The user pressed CTRL+PGDN (only with IMM attribute).
EVENT:Locate	The user pressed the locator VCR button (only with the IMM attribute).
EVENT:ScrollDrag	The user moved the scroll bar's "thumb" and its new position is in PROP:VScrollPos (only with the IMM attribute).
EVENT:ScrollTrack	The user is moving the scroll bar's "thumb" and its new position is in PROP:VScrollPos (only with the IMM attribute).
EVENT:PreAlertKey	The user pressed a printable character (only with the IMM attribute) or an ALRT attribute hot key.
EVENT:AlertKey	The user pressed a printable character (only with the IMM attribute) or an ALRT attribute hot key.
EVENT:Dragging	The mouse cursor is over a potential drag target (only with the DRAGID attribute).
EVENT:Drag	The mouse cursor has been released over a drag target (only with the DRAGID attribute).
EVENT:Drop	The mouse cursor has been released over a drag target (only with the DROPID attribute).
EVENT:DroppingDown	The user has requested the droplist drop down (only with the DROP attribute). CYCLE aborts the dropdown.
EVENT:DroppedDown	The user has dropped the droplist (only with the DROP attribute).
EVENT:Expanding	The user has clicked on a tree expansion box (only with the T in the FORMAT attribute string). CYCLE aborts the expansion.
EVENT:Expanded	The user has clicked on a tree expansion box (only with the T in the FORMAT attribute string).
EVENT:Contracting	The user has clicked on a tree contraction box (only with the T in the FORMAT attribute string). CYCLE aborts the contraction.

EVENT:Contracted The user has clicked on a tree contraction box (only with the T in the FORMAT attribute string).

EVENT:ColumnResize A column in the list has been resized.

Example:

```
Q    QUEUE
F1   STRING(1)
F2   STRING(4)
     END
```

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
        LIST,AT(0,0,20,20),USE(?L1),FROM(Que),IMM
        LIST,AT(20,0,20,20),USE(?L2),FROM(Que),KEY(F10Key)
        LIST,AT(40,0,20,20),USE(?L3),FROM(Que),MSG('Button 3')
        LIST,AT(60,0,20,20),USE(?L4),FROM(Que),HLP('Check4Help')
        LIST,AT(80,0,20,20),USE(?L5),FROM(Q),FORMAT('5C~List~15L~Box~'),COLUMN
        LIST,AT(100,0,20,20),USE(?L6),FROM(Que),FONT('Arial',12)
        LIST,AT(120,0,20,20),USE(?L7),FROM(Que),DROP(6)
        LIST,AT(140,0,20,20),USE(?L8),FROM(Que),HVSCROLL,VCR
        LIST,AT(180,0,20,20),USE(?L10),FROM(Que),CURSOR(CURSOR:Wait)
        LIST,AT(200,0,20,20),USE(?L11),FROM(Que),ALRT(F10Key)
        LIST,AT(220,0,20,20),USE(?L12),FROM(Que),LEFT
        LIST,AT(240,0,20,20),USE(?L13),FROM(Que),RIGHT
        LIST,AT(260,0,20,20),USE(?L14),FROM(Que),CENTER
        LIST,AT(280,0,20,20),USE(?L15),FROM(Que),DECIMAL
        LIST,AT(300,0,20,20),USE(?L16),FROM('Apples|Peaches|Pumpkin|Pie')
        LIST,AT(320,0,20,20),USE(?L17),FROM('TBA')
     END
```

```
CustRpt   REPORT,AT(1000,1000,6500,9000),THOUS
CustDetail DETAIL,AT(0,0,6500,1000)
        LIST,AT(80,0,20,20),USE(?L1),FROM(Q),FORMAT('5C~List~15L~Box~')
     END
     END
CODE
OPEN(MDIChild)
?L1{PROP:From} = 'Live|Long|And|Prosper'    !Runtime FROM attribute assignment
```

See Also:

COMBO

DRAGID

DROPID

SETDROPID

MENU (declare a menu box)

```

MENU(text) [,USE( )] [,KEY( )] [,MSG( )] [,HLP( )] [,STD( )] [,RIGHT] [,DISABLE] [,COLOR] [LEFT( )]
      [,ICON( )] [,FONT( )] [, FIRST | ]
      [, LAST | ]
END

```

MENU	Declares a menu box within a MENUBAR.
<i>text</i>	A string constant containing the display text for the menu selection (PROP:Text).
USE	A field equate label to reference the menu selection in executable code (PROP:USE).
KEY	Specifies an integer constant or keycode equate that immediately opens the menu (PROP:KEY).
MSG	Specifies a string constant containing the text to display in the status bar when the menu is pulled down (PROP:MSG).
HLP	Specifies a string constant containing the help system identifier for the menu (PROP:HLP).
STD	Specifies an integer constant or equate that identifies a "Windows standard behavior" for the menu (PROP:STD).
RIGHT	Specifies the MENU appears at the far right of the action bar (PROP:RIGHT).
DISABLE	Specifies the menu appears dimmed when the WINDOW or APPLICATION is first opened (PROP:DISABLE).
COLOR	Specifies a background color for the control (PROP:COLOR).
LEFT	Specifies the offset in dialog units from the left edge of the menu structure.
ICON	Specifies an image file or standard icon to display on the menu (PROP:ICON).
FONT	Specifies the display font for the control (PROP:FONT).
FIRST	Specifies the MENU appears at the left or top of the menu when merged (PROP:FIRST).
LAST	Specifies the MENU appears at the right or bottom of the menu when merged (PROP:LAST).

MENU declares a drop-down or cascading menu box structure within a MENUBAR structure. When the MENU is selected, the MENU and/or ITEM statements within the structure are displayed in a menu box. A menu box usually appears (drops down) immediately below its *text* on the menu bar (or above, if there is no room below). When selected with ENTER or RIGHT ARROW, any subsequent menu drop-box appears (cascades) immediately to the right of the MENU *text* in the preceding menu box (or left, if there is no room to the right). LEFT ARROW backs up to the preceding menu. The KEY attribute designates a separate accelerator key for the field. This may be any valid Clarion keycode to immediately pull down the MENU.

The *text* string may contain an ampersand (&) which designates the following character as the accelerator "hot" key which is automatically underlined. If the MENU is on the menu bar, pressing the Alt key together with the accelerator key highlights and displays the MENU. If the MENU is within another MENU, pressing the accelerator key, alone, highlights and executes the MENU. If there is no ampersand in the *text*, the first non-blank character in the *text* string is the accelerator key for the MENU, but it will not be underlined. To include an ampersand as part of the *text*, place two ampersands together (&&) in the *text* string and only one will display.

Example:

```
!An MDI application frame window with main menu for the application:
MainWin APPLICATION('My Application'),SYSTEM,MAX,ICON('MyIcon.ICO'),STATUS,HVSCROLL,RES
  MENUBAR
    MENU('File'),USE(?FileMenu),FIRST
      ITEM('Open...'),USE(?OpenFile)
      ITEM('Close'),USE(?CloseFile),DISABLE
      ITEM('E&xit'),USE(?MainExit)
    END
    MENU('Edit'),USE(?EditMenu),KEY(CTRL E),HLP('EditMenuHelp')
      ITEM('Undo'),USE(?UndoText),KEY(CTRL Z),STD(STD:Undo),DISABLE
      ITEM('Cu&t'),USE(?CutText),KEY(CTRL X),STD(STD:Cut),DISABLE
      ITEM('Copy'),USE(?CopyText),KEY(CTRL C),STD(STD:Copy),DISABLE
      ITEM('Paste'),USE(?PasteText),KEY(CTRL V),STD(STD:Paste),DISABLE
      ITEM,SEPARATOR,USE(?FileSeprator1)
    END
    MENU('Window'),STD(STD:WindowList),MSG('Arrange or Select Window'),LAST
      ITEM('Tile'),STD(STD:TileWindow)
      ITEM('Cascade'),STD(STD:CascadeWindow)
      ITEM('Arrange Icons'),STD(STD:ArrangeIcons)
    END
    MENU('Help'),USE(?HelpMenu),LAST,RIGHT
      ITEM('Contents'),USE(?HelpContents),STD(STD:HelpIndex)
      ITEM('Search for Help On...'),USE(?HelpSearch),STD(STD:HelpSearch)
      ITEM('How to Use Help'),USE(?HelpOnHelp),STD(STD:HelpOnHelp)
      ITEM('About MyApp...'),USE(?HelpAbout)
    END
  END
END
```

OLE (declare a window OLE or .OCX container control)

```

OLE ,AT( ) [,CURSOR( )] [,USE( )] [,LAYOUT][,DISABLE] [,KEY( )] [,MSG( )] [,HLP( )] [,SKIP] [,FULL] [,TIP( )]
[,SCROLL] [,ALRT( )] [,HIDE] [,FONT( )] [,DROPID( )] [,COMPATIBILITY( )]
[,CREATE( )] [,CLIP] [,property( value )]
|OPEN( ) |AUTOSIZE |
|LINK( ) |STRETCH |
|DOCUMENT( ) |ZOOM |
[ MENUBAR
  multiple menu and/or item declarations
END ]
END

```

OLE	Places an OLE (Object Linking and Embedding) or .OCX control on the WINDOW or TOOLBAR.
AT	Specifies the initial size and location of the control (PROP:AT). If omitted, default values are selected by the control.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the control (PROP:CURSOR). If omitted, the WINDOW's CURSOR attribute is used, else the Windows default cursor is used.
USE	A Field Equate Label or the label of a variable to receive the "value" of the control (PROP:USE).
LAYOUT	Specifies the control's left-to-right or right-to-left display orientation (PROP:LAYOUT)
DISABLE	Specifies the control appears dimmed when the WINDOW or APPLICATION is first opened (PROP:DISABLE).
KEY	Specifies an integer constant or keycode equate that immediately gives focus to the control (PROP:KEY).
MSG	Specifies a string constant containing the text to display in the status bar when the control has focus (PROP:MSG).
HLP	Specifies a string constant containing the help system identifier for the control (PROP:HLP).
SKIP	Specifies the control does not receive input focus and may only be accessed with the mouse or accelerator key (PROP:SKIP).
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter (PROP:FULL).
TIP	Specifies the text that displays as "balloon help" when the mouse cursor pauses over the control (PROP:ToolTip).
SCROLL	Specifies the control scrolls with the window (PROP:SCROLL).
ALRT	Specifies "hot" keys active for the control (PROP:ALRT).

HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened (PROP:HIDE). UNHIDE must be used to display it.
FONT	Specifies the display font for the control (PROP:FONT).
DROPID	Specifies the control may serve as a drop target for drag-and-drop actions (PROP:DROPID).
COMPATIBILITY	Specifies a compatibility mode for certain OLE or .OCX objects that require it (PROP:COMPATIBILITY).
CREATE	Specifies the control creates a new OLE object or .OCX (PROP:CREATE).
OPEN	Specifies the control opens an object from an OLE Compound Storage file (PROP:AT). When the object is opened, the saved version of the container properties are reloaded, so properties do not need to be re-specified.
LINK	Specifies the OLE object is a link to an object from a file, for example an Excel spreadsheet (PROP:LINK).
DOCUMENT	Specifies the OLE object is an object from a file, for example an Excel spreadsheet (PROP:DOCUMENT).
CLIP	Specifies the OLE object only displays what fits into the size of the OLE container control's AT attribute (PROP:CLIP). If the object is larger than the OLE container control, only the top left corner displays.
AUTOSIZE	Specifies the OLE object automatically resizes itself when the OLE container control's AT attribute parameters change at runtime using PROP:AT, (PROP:AUTOSIZE).
STRETCH	Specifies the OLE object stretches to completely fill the size specified by the OLE container control's AT attribute (PROP:STRETCH).
ZOOM	Specifies the OLE object stretches to fill the size specified by the OLE container control's AT attribute while maintaining the object's aspect ratio (PROP:ZOOM).
<i>property</i>	A string constant containing the name of a custom property setting for the control.
<i>value</i>	A string constant containing the property value number or EQUATE for the <i>property</i> .
MENUBAR	Defines a menu structure for the control. This is exactly the same type of structure as a MENUBAR in an APPLICATION or WINDOW structure and is merged into the application's menu.
<i>menus and/or items</i>	MENU and/or ITEM declarations that define the menu selections.

The **OLE** control places an OLE or .OCX control on the WINDOW or TOOLBAR (not valid in a REPORT) at the position and size specified by its AT attribute. The *property* attribute allows you to specify any additional property settings the OLE or .OCX control may require. These are properties that need to be set for the OLE or .OCX control to properly function, and are not standard Clarion properties (such as AT, CURSOR, or USE). The custom control should only receive values for these properties that are defined for that control. Valid properties and values for those properties would be defined in the custom control's documentation. You may have multiple *property* attributes on a single OLE control.

Events Generated:

EVENT:Selected	The control has received input focus.
EVENT:Accepted	The user has completed using the control.
EVENT:PreAlertKey	The user pressed an ALRT attribute hot key.
EVENT:AlertKey	The user pressed an ALRT attribute hot key.
EVENT:Drop	A successful drag-and-drop to the control.

Example:

```

PROGRAM

MAP
  INCLUDE('OCX.CLW')
END

W WINDOW('OCX Controls'),AT(,,200,200),RESIZE,STATUS(-1,-1),SYSTEM
  MENUBAR
    ITEM('E&xit!'),USE(?Exit)
    ITEM('&About!'),USE(?About)
    ITEM('&Properties!'),USE(?Property)
  END
  OLE,AT(0,0,0,0),USE(?oc1),HIDE,CREATE('COMCTL.ImagelistCtrl.1').
  OLE,AT(0,0,150,20),USE(?oc2),CREATE('TOOLBAR.ToolbarCtrl.1').
  END

CODE
OPEN(W)
?oc1{'ListImages.Add(1,xyz,' & ocxloadimage('IRCLOCK.BMP') & ')'}
?oc1{'ListImages.Add(2,abc,' & ocxloadimage('IRCLOCK2.BMP') & ')'}
?oc2{'ImageList'} = ?oc1{PROP:Object}
LOOP X# = 1 TO 3
  ?oc2{'Buttons.Add(,,,1)'}; ?oc2{'Buttons.Add(,,,2)'}
END
ACCEPT
CASE EVENT()
OF EVENT:Accepted

```

```
CASE FIELD()  
OF ?Exit  
  BREAK  
OF ?About  
  ?oc1{'AboutBox'}           !Display the OCX control's About Box  
OF ?Property  
  ?oc1{PROP:DoVerb} = -7    !Display the OCX control's properties dialog  
END  
END  
END
```

See Also:

Object Linking and Embedding

OLE (.OCX) Custom Controls

OCX Library Procedures

OPTION (declare a set of RADIO controls)

```
OPTION(text ,AT( ) [.CURSOR( )] [.USE( )] [.LAYOUT( )][.DISABLE] [.KEY( )] [.MSG( )] [.HLP( )] [.BOXED]
[.FULL] [.SCROLL] [.HIDE] [.FONT( )] [.ALRT( )] [.SKIP] [DROPID( )] [.TIP( )] [.TRN] [.COLOR( )]
[.BEVEL( )] [.EXTEND( )]
```

radios

END

OPTION	Declares a set of RADIO controls.
<i>text</i>	A string constant containing the prompt for the set of controls (PROP:Text). This may contain an ampersand (&) to indicate the "hot" letter for the prompt. The <i>text</i> is displayed on screen only if the BOXED attribute is also present.
AT	Specifies the initial size and location of the control (PROP:AT). If omitted, default values are selected by the runtime library.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the control (PROP:CURSOR). If omitted, the WINDOW's CURSOR attribute is used, else the Windows default cursor is used. Not valid in a REPORT.
USE	The label of a variable to receive the choice (PROP:USE). If this is a string variable, it receives the value of the RADIO string (with any accelerator key ampersand stripped out) selected by the user. If a numeric variable, it receives the ordinal position within the OPTION of the RADIO button selected by the user (the value returned by the CHOICE() procedure).
LAYOUT	Specifies the control's left-to-right or right-to-left display orientation (PROP:LAYOUT)
DISABLE	Specifies the control appears dimmed when the WINDOW or APPLICATION is first opened (PROP:DISABLE).
KEY	Specifies an integer constant or keycode equate that immediately gives focus to the currently selected RADIO in the OPTION control (PROP:KEY). Not valid in a REPORT.
MSG	Specifies a string constant containing the default text to display in the status bar when any control in the OPTION has focus (PROP:MSG). Not valid in a REPORT.
HLP	Specifies a string constant containing the default help system identifier for any control in the OPTION (PROP:HLP). Not valid in a REPORT.
BOXED	Specifies a single-track border around the RADIO controls with the text at the top of the border (PROP:BOXED).

FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter (PROP:FULL). Not valid in a REPORT.
SCROLL	Specifies the control scrolls with the window (PROP:SCROLL). Not valid in a REPORT.
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened (PROP:HIDE). UNHIDE must be used to display it.
FONT	Specifies the display font for the control and the default for all the controls in the OPTION (PROP:FONT).
ALRT	Specifies "hot" keys active for the controls in the OPTION (PROP:ALRT). Not valid in a REPORT.
SKIP	Specifies the controls in the OPTION do not receive input focus and may only be accessed with the mouse or accelerator key (PROP:SKIP). Not valid in a REPORT.
DROPID	Specifies the control may serve as a drop target for drag-and-drop actions (PROP:DROPID). Not valid in a REPORT.
TIP	Specifies the text that displays as "balloon help" when the mouse cursor pauses over the control (PROP:ToolTip). Not valid in a REPORT.
COLOR	Specifies a background color for the control (PROP:COLOR).
BEVEL	Specifies custom 3-D effect borders (PROP:BEVEL). Not valid in a REPORT.
TRN	Specifies the text or USE variable characters transparently display over the background (PROP:Trn).
EXTEND	Specifies special document formatting information. This information contains a valid string of attributes that are assigned to a designated REPORT control for a given document type. Valid only in a REPORT.

radios Multiple RADIO control declarations.

The **OPTION** control declares a set of RADIO controls which offer the user a list of choices. The multiple RADIO controls in the OPTION structure define the choices offered to the user. On a REPORT, the OPTION control prints a group of RADIO controls which display a list of choices. The selected choice is identified by a filled RADIO button.

Input focus changes between the OPTION's RADIO controls are signalled only to the individual RADIO controls affected. This means the EVENT:Selected events generated when the user changes input focus within an OPTION structure are field-specific events for the affected RADIO controls, not the OPTION structure which contains them. There is no EVENT:Selected generated for an OPTION structure. However, the RADIO control does not receive EVENT:Accepted, the OPTION structure receives the EVENT:Accepted when the user has selected a RADIO.

A string variable as the OPTION structure's USE attribute receives the text of the RADIO control selected by the user, and the CHOICE(?Option) procedure returns the number of the selected RADIO button. If the contents of the OPTION structure's USE attribute is a numeric variable, it receives the number of the RADIO button selected by the user (the value returned by the CHOICE procedure).

No RADIO button selected is a valid option, which occurs only when the OPTION structure's USE variable does not contain a value related to one of its component RADIO controls. This condition only lasts until the user has selected one of the RADIOS.

Events Generated:

EVENT:Accepted One of the OPTION's RADIO controls has been selected by the user.
 EVENT:PreAlertKey The user pressed an ALRT attribute hot key.
 EVENT:AlertKey The user pressed an ALRT attribute hot key.
 EVENT:Drop A successful drag-and-drop to the control.

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
  OPTION('Option 1'),USE(OptVar1),KEY(F10Key),HLP('Option1Help')
  RADIO('Radio 1'),AT(0,0,20,20),USE(?R1)
  RADIO('Radio 2'),AT(20,0,20,20),USE(?R2)
  END
  OPTION('Option 2'),USE(OptVar2),MSG('Option 2'),SCROLL
  RADIO('Radio 3'),AT(40,0,20,20),USE(?R3)
  RADIO('Radio 4'),AT(60,0,20,20),USE(?R4)
  END
  OPTION('Option 3'),USE(OptVar3),AT(80,0,20,20),BOXED
  RADIO('Radio 5'),AT(80,0,20,20),USE(?R5)
  RADIO('Radio 6'),AT(100,0,20,20),USE(?R6)
  END
  OPTION('Option 4'),USE(OptVar4),FONT('Arial',12),CURSOR(CURSOR:Wait)
  RADIO('Radio 7'),AT(120,0,20,20),USE(?R7)
  RADIO('Radio 8'),AT(140,0,20,20),USE(?R8)
  END
  END
CustRpt REPORT,AT(1000,1000,6500,9000),THOUS
CustDetail DETAIL,AT(0,0,6500,1000)
  OPTION('Option'),USE(OptVar),AT(80,0,20,20),BOXED
  RADIO('Radio 1'),AT(80,0,20,20),USE(?R1)
  RADIO('Radio 2'),AT(100,0,20,20),USE(?R2)
  END
  END
  END
```

See Also: RADIO, BUTTON, CHECK

PANEL (declare a panel control)

PANEL ,AT() [,USE()] [,LAYOUT()][,DISABLE] [,FULL] [,FILL()] [,SCROLL] [,HIDE] [,BEVEL()]

PANEL	Defines an area in the WINDOW or TOOLBAR.
AT	Specifies the initial size and location of the control (PROP:AT). If omitted, default values are selected by the runtime library.
USE	A field equate label to reference the control in executable code (PROP:USE).
LAYOUT	Specifies the control's left-to-right or right-to-left display orientation (PROP:LAYOUT)
DISABLE	Specifies the control is disabled when the WINDOW or APPLICATION is first opened (PROP:DISABLE).
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter (PROP:FULL).
FILL	Specifies the fill color for the control (PROP:FILL). If omitted, the panel is not filled with color.
SCROLL	Specifies the control scrolls with the window (PROP:SCROLL).
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened (PROP:HIDE). UNHIDE must be used to display it.
BEVEL	Specifies custom 3-D effect borders (PROP:BEVEL).

The **PANEL** control defines an area WINDOW or TOOLBAR (not valid in a REPORT) at the position and size specified by its AT attribute. Typically, the purpose of a PANEL is to frame the area with a custom BEVEL. This control cannot receive input focus and does not generate events.

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
    PANEL,AT(10,100,20,20),USE(?P1),BEVEL(-2,2)
END
```

See Also:

BOX

GROUP

PROMPT (declare a prompt control)

```
PROMPT(text) ,AT( ) [,CURSOR( )] [,USE( )] [,LAYOUT( )][,DISABLE] [,FONT( )] [,FULL] [,SCROLL] [,TRN]
      [,HIDE] [,DROPID( )] [,
      | LEFT | ] [,COLOR( )]
      | RIGHT |
      | CENTER |
```

PROMPT	Places a prompt for the next active control following it, in the WINDOW or TOOLBAR.
<i>text</i>	A string constant containing the text to display (PROP:Text). This may contain an ampersand (&) to indicate the "hot" letter for the prompt.
AT	Specifies the initial size and location of the control (PROP:AT). If omitted, default values are selected by the runtime library.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the control (PROP:CURSOR). If omitted, the WINDOW's CURSOR attribute is used, else the Windows default cursor is used.
USE	A field equate label to reference the control in executable code (PROP:USE).
LAYOUT	Specifies the control's left-to-right or right-to-left display orientation (PROP:LAYOUT)
DISABLE	Specifies the control appears dimmed when the WINDOW or APPLICATION is first opened (PROP:DISABLE).
FONT	Specifies the font used to display the text (PROP:FONT).
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter (PROP:FULL).
SCROLL	Specifies the control scrolls with the window (PROP:SCROLL).
TRN	Specifies the control transparently displays over the background (PROP:TRN).
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened (PROP:HIDE). UNHIDE must be used to display it.
DROPID	Specifies the control may serve as a drop target for drag-and-drop actions (PROP:DROPID).
LEFT	Specifies that the prompt is left justified (PROP:LEFT).
RIGHT	Specifies that the prompt is right justified (PROP:RIGHT).
CENTER	Specifies that the prompt is centered (PROP:CENTER).
COLOR	Specifies a background color for the control (PROP:COLOR).

The **PROMPT** control places a prompt for the next active control following the PROMPT in the WINDOW or TOOLBAR structure (not valid in a REPORT). The prompt *text* is placed at the position and size specified by its AT attribute.

The *text* may contain an ampersand (&) to indicate the letter immediately following the ampersand is the "hot" letter for the prompt. By default, the "hot" letter displays with an underscore below it to indicate its special purpose. This "hot" letter, when pressed in conjunction with the ALT key, changes input focus to the next control following the PROMPT in the WINDOW or TOOLBAR structure, which is capable of receiving focus.

Disabling or hiding the control directly following the PROMPT in the window structure does not automatically disable or hide the PROMPT; it must also be explicitly disabled or hidden, otherwise the PROMPT will then refer to the next currently active control following the disabled control. This allows you to place one PROMPT control on the window that will apply to any of multiple controls (if only one will be active at a time). If the next active control is a BUTTON, it is pressed when the user presses the PROMPT's "hot key."

To include an ampersand as part of the prompt *text*, place two ampersands together (&&) in the *text* string and only one will display.

This control cannot receive input focus.

Events Generated:

EVENT:Drop A successful drag-and-drop to the control.

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
    PROMPT('Enter Data:'),AT(10,100,20,20),USE(?P1),CURSOR(CURSOR:Wait)
    ENTRY(@S8),AT(100,100,20,20),USE(E1)
    PROMPT('Enter More Data:'),AT(10,200,20,20),USE(?P2),CURSOR(CURSOR:Wait)
    ENTRY(@S8),AT(100,200,20,20),USE(E2)
    ENTRY(@D1),AT(100,200,20,20),USE(E3)
END

CODE
OPEN(MDIChild)
IF SomeCondition
    HIDE(?E2)        !Prompt will refer to E3
ELSE
    HIDE(?E3)        !Prompt will refer to E2
END
```

See Also:

ENTRY

TEXT

PROGRESS (declare a progress control)

PROGRESS, **AT**() [, **CURSOR**()] [, **USE**()] [, **LAYOUT**()][, **DISABLE**] [, **FULL**] [, **SCROLL**] [, **HIDE**]
[, **DROPID**()][, **RANGE**()]

PROGRESS	Places a control that displays the current progress of a batch process in the WINDOW or TOOLBAR.
AT	Specifies the initial size and location of the control (PROP:AT). If omitted, default values are selected by the runtime library.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the control (PROP:CURSOR). If omitted, the WINDOW's CURSOR attribute is used, else the Windows default cursor is used.
USE	The label of the variable containing the value of the current progress, or a field equate label to reference the control in executable code (PROP:USE).
LAYOUT	Specifies the control's left-to-right or right-to-left display orientation (PROP:LAYOUT)
DISABLE	Specifies the control appears dimmed when the WINDOW or APPLICATION is first opened (PROP:DISABLE).
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter (PROP:FULL).
SCROLL	Specifies the control scrolls with the window (PROP:SCROLL).
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened (PROP:HIDE). UNHIDE must be used to display it.
DROPID	Specifies the control may serve as a drop target for drag-and-drop actions (PROP:DROPID).
RANGE	Specifies the range of values the progress bar displays (PROP:RANGE). If omitted, the default range is zero (0) to one hundred (100).

The **PROGRESS** control declares a control that displays a progress bar in a WINDOW or TOOLBAR (not valid in a REPORT). This usually displays the current percentage of completion of a batch process.

If a variable is named as the USE attribute, the progress bar is automatically updated whenever the value in that variable changes. If the USE attribute is a field equate label, you must directly update the display by assigning a value (within the range defined by the RANGE attribute) to the control's PROP:progress property (an undeclared property equate -- see *Undeclared Properties*).

This control cannot receive input focus.

Events Generated:

EVENT:Drop A successful drag-and-drop to the control.

Example:

```
BackgroundProcess PROCEDURE       !Background processing batch process

ProgressVariable LONG

Win WINDOW('Batch Processing...'),AT(, ,400,400),TIMER(1),MDI,CENTER
    PROGRESS,AT(100,100,200,20),USE(ProgressVariable),RANGE(0,200)
    PROGRESS,AT(100,140,200,20),USE(?ProgressBar),RANGE(0,200)
    BUTTON('Cancel'),AT(190,300,20,20),STD(STD:Close)
    END

CODE
OPEN(Win)
OPEN(File)
?ProgressVariable{PROP:rangehigh} = RECORDS(File)
?ProgressBar{PROP:rangehigh} = RECORDS(File)
SET(File)                         !Set up a batch process
ACCEPT
CASE EVENT()
OF EVENT:CloseWindow
    BREAK
OF EVENT:Timer                    !Process records when timer allows it
    ProgressVariable += 3         !Auto-updates 1st progress bar
    LOOP 3 TIMES
    NEXT(File)
    IF ERRORCODE()
    BREAK
    END
    ?ProgressBar{PROP:progress} = ?ProgressBar{PROP:progress} + 1
                                  !Manually update 2nd progress bar
    !Perform some batch processing code HERE
    END
END
END
END
CLOSE(File)
```

RADIO (declare a radio button control)

```

RADIO(text) ,AT( ) [,CURSOR( )] [,USE( )] [,LAYOUT( )] [,DISABLE] [,KEY( )] [,MSG( )] [,HLP( )] [,SKIP]
      [,FONT( )] [,ICON( )] [,FULL] [,SCROLL] [,HIDE] [,ALRT( )] [DROPID( )] [VALUE( )]
      [,TIP( )] [,TRN] [,COLOR( )] [,FLAT] [,
      | LEFT | ] [,EXTEND( )]
      | RIGHT |

```

RADIO	Places a radio button on the WINDOW or TOOLBAR.
<i>text</i>	A string constant containing the text to display for the radio button (PROP:Text). This may contain an ampersand (&) to indicate the "hot" letter for the radio button.
AT	Specifies the initial size and location of the control (PROP:AT). If omitted, default values are selected by the runtime library.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the control (PROP:CURSOR). If omitted, the WINDOW's CURSOR attribute is used, else the Windows default cursor is used. Not valid in a REPORT.
USE	A field equate label to reference the control in executable code (PROP:USE).
LAYOUT	Specifies the control's left-to-right or right-to-left display orientation (PROP:LAYOUT)
DISABLE	Specifies the control appears dimmed when the WINDOW or APPLICATION opens (PROP:DISABLE).
KEY	Specifies an integer constant or keycode equate that immediately selects the radio button (PROP:KEY). Not valid in a REPORT.
MSG	Specifies a string constant containing the text to display in the status bar when the control has focus (PROP:MSG). Not valid in a REPORT.
HLP	Specifies a string constant containing the help system identifier for the control (PROP:HLP). Not valid in a REPORT.
SKIP	Specifies the control does not receive input focus and may only be accessed with the mouse or accelerator key (PROP:SKIP). Not valid in a REPORT.
FONT	Specifies the display font for the control (PROP:FONT).
ICON	Specifies an image file or standard icon to display on the face of a "latching" button (PROP:ICON). Not valid in a REPORT.
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter (PROP:FULL). Not valid in a REPORT.
SCROLL	Specifies the control scrolls with the window (PROP:SCROLL). Not valid in a REPORT.

HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened (PROP:HIDE). UNHIDE must be used to display it.
ALRT	Specifies "hot" keys active for the control (PROP:ALRT). Not valid in a REPORT.
DROPID	Specifies the control may serve as a drop target for drag-and-drop actions (PROP:DROPID). Not valid in a REPORT.
VALUE	Specifies the value the OPTION structure's USE variable receives when the radio button is selected by the user (PROP:VALUE).
TIP	Specifies the text that displays as "balloon help" when the mouse cursor pauses over the control (PROP:ToolTip). Not valid in a REPORT.
TRN	Specifies the control transparently displays over the background (PROP:TRN).
COLOR	Specifies a background color for the control (PROP:COLOR).
FLAT	Specifies the button appears flat except when the mouse cursor passes over the control (PROP:FLAT). Requires the ICON attribute. Not valid in a REPORT.
LEFT	Specifies the text appears to the left of the radio button (PROP:LEFT).
RIGHT	Specifies the text appears to the right of the radio button (PROP:RIGHT). This is the default position.
EXTEND	Specifies special document formatting information. This information contains a valid string of attributes that are assigned to a designated REPORT control for a given document type. Valid only in a REPORT.

The **RADIO** control places a radio button on the WINDOW, TOOLBAR, or REPORT at the position and size specified by its AT attribute. A RADIO control may only be placed within an OPTION control. When selected by the user, the RADIO *text* (with any accelerator key ampersand stripped out) is placed in the OPTION's USE variable, unless the VALUE attribute is used. On a REPORT, the RADIO selected by the user (the value in the OPTION's USE variable) is displayed as a filled RADIO button.

A RADIO with an ICON attribute appears as a "latched" pushbutton with the image on the button face. When the button appears "up" the RADIO is off; when it appears "down" the RADIO is on and the OPTION's USE variable receives the value in the selected RADIO's *text* parameter (unless the VALUE attribute is used).

There is an EVENT:Selected is generated for a RADIO control, but the OPTION structure containing it receives the EVENT:Accepted.

Events Generated:

EVENT:Selected	The control has received input focus.
EVENT:PreAlertKey	The user pressed an ALRT attribute hot key.
EVENT:AlertKey	The user pressed an ALRT attribute hot key.
EVENT:Drop	A successful drag-and-drop to the control.

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
  OPTION('Option 1'),USE(OptVar1)
  RADIO('Radio 1'),AT(0,0,20,20),USE(?R1),KEY(F10Key)
  RADIO('Radio 2'),AT(20,0,20,20),USE(?R2),MSG('Radio 2')
  END
  OPTION('Option 2'),USE(OptVar2)
  RADIO('Radio 3'),AT(40,0,20,20),USE(?R3),FONT('Arial',12)
  RADIO('Radio 4'),AT(60,0,20,20),USE(?R4),CURSOR(CURSOR:Wait)
  END
  OPTION('Option 3'),USE(OptVar3)
  RADIO('Radio 5'),AT(80,0,20,20),USE(?R5),HLP('Radio5Help')
  RADIO('Radio 6'),AT(100,0,20,20),USE(?R6)
  END
  OPTION('Option 4'),USE(OptVar4)
  RADIO('Radio 7'),AT(120,0,20,20),USE(?R7),ICON('Radio1.ICO')
  RADIO('Radio 8'),AT(140,0,20,20),USE(?R8),ICON('Radio2.ICO')
  END
  OPTION('Option 5'),USE(OptVar5)
  RADIO('Radio 9'),AT(100,20,20,20),USE(?R9),LEFT
  RADIO('Radio 10'),AT(120,20,20,20),USE(?R10),LEFT
  END
  OPTION('Option 6'),USE(OptVar6),SCROLL
  RADIO('Radio 11'),AT(200,0,20,20),USE(?R11),SCROLL
  RADIO('Radio 12'),AT(220,0,20,20),USE(?R12),SCROLL
  END
END

CustRpt   REPORT,AT(1000,1000,6500,9000),THOUS
CustDetail DETAIL,AT(0,0,6500,1000)
  OPTION('Option'),USE(OptVar),AT(80,0,20,20),BOXED
  RADIO('Radio 1'),AT(80,0,20,20),USE(?R1)
  RADIO('Radio 2'),AT(100,0,20,20),USE(?R2)
  RADIO('Radio 3'),AT(100,0,20,20),USE(?R2),LEFT
  END
  END
  END
```

See Also:

OPTION

CHECK

BUTTON

REGION (declare a window region control)

REGION ,**AT**() [,**CURSOR**()] [,**USE**()] [,**LAYOUT**()] [,**DISABLE**] [,**FILL**] [,**COLOR**()] [,**IMM**] [,**FULL**] [,**TRN**] [,**SCROLL**] [,**HIDE**] [,**DRAGID**()] [,**DROPID**()] [,**BEVEL**()]

REGION	Defines an area in the WINDOW or TOOLBAR.
AT	Specifies the initial size and location of the control (PROP:AT). If omitted, default values are selected by the runtime library.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the control (PROP:CURSOR). If omitted, the WINDOW's CURSOR attribute is used, else the Windows default cursor is used.
USE	A field equate label to reference the control in executable code (PROP:USE).
LAYOUT	Specifies the control's left-to-right or right-to-left display orientation (PROP:LAYOUT)
DISABLE	Specifies the control is disabled when the WINDOW or APPLICATION is first opened (PROP:DISABLE).
FILL	Specifies the red, green, and blue component values that create the fill color for the control (PROP:FILL). If omitted, the region is not filled with color.
COLOR	Specifies the border color of the control (PROP:COLOR). If omitted, there is no border.
IMM	Specifies control generates an event whenever the mouse is moved in the region (PROP:IMM).
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter (PROP:FULL).
SCROLL	Specifies the control scrolls with the window (PROP:SCROLL).
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened (PROP:HIDE). UNHIDE must be used to display it.
DRAGID	Specifies the control may serve as a drag host for drag-and-drop actions (PROP:DRAGID).
DROPID	Specifies the control may serve as a drop target for drag-and-drop actions (PROP:DROPID).
BEVEL	Specifies custom 3-D effect borders (PROP:BEVEL).
TRN	Specifies the control transparently displays over the background (PROP:TRN).

The **REGION** control defines an area on a WINDOW or TOOLBAR (not valid in a REPORT) at the position and size specified by its AT attribute. Generally, tracking the position of the mouse is the reason for defining a REGION. The MOUSEX and MOUSEY procedures can be used to determine the exact position of the mouse when the event occurs. Use of the IMM attribute causes some excess code and speed overhead at runtime, so it should be used only when necessary. This control cannot receive input focus.

A REGION with the DRAGID attribute can serve as a drag-and-drop host, providing information to be moved or copied to another control. A REGION with the DROPID attribute can serve as a drag-and-drop target, receiving information from another control. These attributes work together to specify drag-and-drop "signatures" that define a valid target for the operation. The DRAGID() and DROPID() procedures, along with the SETDROPID procedure, are used to perform the data exchange. Since a REGION can be defined over any other control, you can write drag-and-drop code between any two controls. Simply define REGION controls to handle the required drag-and-drop functionality.

Events Generated:

EVENT:Accepted	The mouse has been clicked by the user in the region.
EVENT:MouseIn	The mouse has entered the region (only with the IMM attribute).
EVENT:MouseOut	The mouse has left the region (only with the IMM attribute).
EVENT:MouseMove	The mouse has moved within the region (only with the IMM attribute).
EVENT:Dragging	The mouse cursor is over a potential drag target (only with the DRAGID attribute).
EVENT:Drag	The mouse cursor has been released over a drag target (only with the DRAGID attribute).
EVENT:Drop	The mouse cursor has been released over a drag target (only with the DROPID attribute).

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
    REGION,AT(10,100,20,20),USE(?R1),BEVEL(-2,2)
    REGION,AT(100,100,20,20),USE(?R2),CURSOR(CURSOR:Wait)
    REGION,AT(10,200,20,20),USE(?R3),IMM
    REGION,AT(100,200,20,20),USE(?R4),COLOR(COLOR:ACTIVEBORDER)
    REGION,AT(10,300,20,20),USE(?R5),FILL(COLOR:ACTIVEBORDER)
END
```

See Also:

PANEL

SHEET (declare a group of TAB controls)

```

SHEET ,AT( ) [,CURSOR( )][,USE( )] [,LAYOUT( )] [,DISABLE][,KEY( )][,FULL][,SCROLL][,HIDE][,FONT()]
      [,DROPID()][,WIZARD][,SPREAD][,HSCROLL][,JOIN][,NOSHEET][,COLOR()]
      [,UP ] [,DOWN ] [, LEFT  ] [, IMM ]
                        | RIGHT |
                        | ABOVE  |
                        | BELOW  |
      tabs
      END

```

SHEET	Declares a group of TAB controls.
AT	Specifies the initial size and location of the control (PROP:AT). If omitted, default values are selected by the runtime library.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the control (PROP:CURSOR). If omitted, the WINDOW's CURSOR attribute is used, else the Windows default cursor is used.
USE	The label of a variable to receive the current TAB choice (PROP:USE). If this is a string variable, it receives the value of the TAB string (with any ampersands stripped out) selected by the user. If a numeric variable, it receives the number of the TAB selected by the user (the value returned by the CHOICE() procedure).
LAYOUT	Specifies the control's left-to-right or right-to-left display orientation (PROP:LAYOUT)
DISABLE	Specifies the control appears dimmed when the WINDOW or APPLICATION opens (PROP:DISABLE).
KEY	Specifies an integer constant or keycode equate that immediately gives focus to the currently selected TAB in the SHEET control (PROP:KEY).
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter (PROP:FULL).
SCROLL	Specifies the control scrolls with the window (PROP:SCROLL).
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened (PROP:HIDE). UNHIDE must be used to display it.
FONT	Specifies the display font for the control and the default for all the controls in the SHEET (PROP:FONT).
DROPID	Specifies the control may serve as a drop target for drag-and-drop actions (PROP:DROPID).
WIZARD	Specifies the SHEET's TAB controls do not appear (PROP:WIZARD). The user moves from TAB to TAB under program control.
SPREAD	Specifies the TABs are evenly spaced on one line (PROP:SPREAD).

HSCROLL	Specifies the TABs display all on one row instead of multiple rows, no matter how many TABs there are (PROP:HSCROLL). Right and left (or up and down) scroll buttons appear at either end of the TABs to scroll through the TABs.
JOIN	Specifies the TABs display all on one row instead of multiple rows, no matter how many TABs there are (PROP:JOIN). Right and left (or up and down) scroll buttons appear together at the right (or bottom) end of the TABs to scroll through the TABs.
NOSHEET	Specifies the TABs display without a visible sheet (PROP:NOSHEET).
COLOR	Specifies a background color for the control (PROP:COLOR).
UP	Specifies the TAB text is vertical reading upwards (PROP:UP).
DOWN	Specifies the TAB text is vertical reading downwards (PROP:DOWN).
LEFT	Specifies the TABs appear to the left of the sheet (PROP:LEFT).
RIGHT	Specifies the TABs appear to the right of the sheet (PROP:RIGHT).
ABOVE	Specifies the TABs appear above the sheet (PROP:ABOVE). This is the default position.
BELOW	Specifies the TABs appear below the sheet (PROP:BELOW).
IMM	Specifies EVENT:NewSelection generates whenever the user clicks on a TAB (PROP:IMM).
<i>tabs</i>	Multiple TAB control declarations.

The **SHEET** control declares a group of TAB controls that offer the user multiple "pages" of controls for the window (not valid in a REPORT). The TAB controls in the SHEET structure define the "pages" displayed.

Input focus changes between the SHEET's TAB controls are signalled only to the SHEET control affected. This means the events generated when the user changes input focus within a SHEET structure are field-specific events for the affected SHEET structure, not the individual TAB control.

A string variable as the SHEET structure's USE attribute receives the text of the TAB control selected by the user, and the CHOICE(*?Option*) procedure returns the number of the selected TAB control. If the SHEET structure's USE attribute is a numeric variable, it receives the number of the TAB control selected by the user (the same value returned by the CHOICE procedure).

You can use the SELECT statement to force navigation to a specific tab by specifying the TAB control's position number within the sheet as the second parameter:
SELECT(?Sheet,TabNumber).

Events Generated:

- EVENT:TabChanging Focus is about to pass to another tab.
- EVENT:NewSelection Focus has passed to another tab, or the user clicked on a TAB in a SHEET with the IMM attribute.
- EVENT:Drop A successful drag-and-drop to the control.

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
SHEET,AT(0,0,320,175),USE(selectedTab)
TAB('Tab One'),USE(?TabOne)
OPTION('Option 1'),USE(OptVar1),KEY(F10Key),HLP('Option1Help')
RADIO('Radio 1'),AT(20,0,20,20),USE(?R1)
RADIO('Radio 2'),AT(40,0,20,20),USE(?R2)
END
OPTION('Option 2'),USE(OptVar2),MSG('Option 2')
RADIO('Radio 3'),AT(60,0,20,20),USE(?R3)
RADIO('Radio 4'),AT(80,0,20,20),USE(?R4)
END
PROMPT('Enter Data:'),AT(100,100,20,20),USE(?P1)
ENTRY(@S8),AT(100,140,32,20),USE(E1)
PROMPT('Enter More Data:'),AT(100,200,20,20),USE(?P2)
ENTRY(@S8),AT(100,240,32,20),USE(E2)
END
TAB('Tab Two'),USE(?TabTwo)
OPTION('Option 3'),USE(OptVar3)
RADIO('Radio 1'),AT(20,0,20,20),USE(?R5)
RADIO('Radio 2'),AT(40,0,20,20),USE(?R6)
END
PROMPT('Enter Data:'),AT(100,100,20,20),USE(?P3)
ENTRY(@S8),AT(100,140,32,20),USE(E3)
PROMPT('Enter More Data:'),AT(100,200,20,20),USE(?P4)
ENTRY(@S8),AT(100,240,32,20),USE(E4)
END
END
BUTTON('Ok'),AT(100,180,20,20),USE(?Ok)
BUTTON('Cancel'),AT(200,180,20,20),USE(?Cancel)
END
```

See Also:

TAB

SPIN (declare a spinning list control)

```

SPIN(picture) ,AT( ) [,CURSOR()] [,USE()] [,LAYOUT( )] [,DISABLE] [,KEY()] [,MSG()] [,HLP()] [,SKIP] [,FONT()]
[,FULL][,SCROLL][,ALRT()][,HIDE][,READONLY][,REQ][,IMM][,TIP()][,TRN][,DROPID()][,COLOR()]
[,REPEAT( )] [,DELAY( )] [,MASK]
[,|UPR |][,|LEFT |][,|INS |][,|RANGE()[,STEP |][,|HSCROLL|]
|CAP | |RIGHT | |OVR | |FROM( ) | |VSCROLL|
|CENTER |
|DECIMAL |

```

SPIN	Places a "spinning" list of data items on the WINDOW or TOOLBAR.
<i>picture</i>	A display picture token that specifies the format for the data displayed in the control (PROP:Text).
AT	Specifies the initial size and location of the control (PROP:AT). If omitted, default values are selected by the runtime library.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the control (PROP:CURSOR). If omitted, the WINDOW's CURSOR attribute is used, else the Windows default cursor is used.
USE	A field equate label to reference the control in executable code or the label of the variable that receives the value selected by the user (PROP:USE).
LAYOUT	Specifies the control's left-to-right or right-to-left display orientation (PROP:LAYOUT)
DISABLE	Specifies the control appears dimmed when the WINDOW or APPLICATION opens (PROP:DISABLE).
KEY	Specifies an integer constant or keycode equate that immediately gives focus to the control (PROP:KEY).
MSG	Specifies a string constant containing the text to display in the status bar when the control has focus (PROP:MSG).
HLP	Specifies a string constant containing the help system identifier for the control (PROP:HLP).
SKIP	Specifies the control receives input focus to enter text only with the mouse or accelerator key and does not retain focus (PROP:SKIP).
FONT	Specifies the display font for the control (PROP:FONT).
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter (PROP:FULL).
SCROLL	Specifies the control scrolls with the window (PROP:SCROLL).
ALRT	Specifies "hot" keys active for the control (PROP:ALRT).
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened (PROP:HIDE). UNHIDE must be used to display it.

READONLY	Specifies the control does not allow data entry (PROP:READONLY).
REQ	Specifies the control may not be left blank or zero (PROP:REQ).
IMM	Specifies immediate event generation whenever the user presses any key (PROP:IMM).
TIP	Specifies the text that displays as "balloon help" when the mouse cursor pauses over the control (PROP:ToolTip).
TRN	Specifies the control transparently displays over the background (PROP:TRN).
DROPID	Specifies the control may serve as a drop target for drag-and-drop actions (PROP:DROPID).
COLOR	Specifies background and selected colors for the control (PROP:COLOR).
REPEAT	Specifies the rate at which EVENT:NewSelection generates when the spin buttons are held down by the user (PROP:REPEAT).
DELAY	Specifies the delay between the first and second generation of EVENT:NewSelection when the spin buttons are held down by the user (PROP:DELAY).
MASK	Specifies pattern input editing mode of the ENTRY portion of the control (PROP:MASK).
UPR / CAP	Specifies all upper case or proper name capitalization (First Letter Of Each Word Capitalized) entry (PROP:UPR and PROP:CAP).
LEFT	Specifies that the data is left justified within the area specified by the AT attribute (PROP:LEFT).
RIGHT	Specifies that the data is right justified within the area specified by the AT attribute (PROP:RIGHT).
CENTER	Specifies that the data is centered within the area specified by the AT attribute (PROP:CENTER).
DECIMAL	Specifies that the data is aligned on the decimal point within the area specified by the AT attribute (PROP:DECIMAL).
INS / OVR	Specifies Insert or Overwrite entry mode (PROP:INS and PROP:OVR). Valid only on windows with the MASK attribute.
RANGE	Specifies the range of values the user may choose (PROP:RANGE).
STEP	Specifies the increment/decrement amount of the choices within the specified RANGE (PROP:STEP). If omitted, the STEP is 1.0.
FROM	Specifies the origin of the choices displayed for the user (PROP:FROM).
HSCROLL	Specifies the spin buttons are side by side, pointing right and left (PROP:HSCROLL).

- VSCROLL** Specifies the spin buttons are one above the other, pointing right and left (PROP:VSCROLL).
- HVSCROLL** Specifies the spin buttons are side by side, pointing up and down (PROP:HVSCROLL).

The **SPIN** control places a "spinning" list of data items on the WINDOW or TOOLBAR (not valid in a REPORT) at the position and size specified by its AT attribute. The "spinning" list displays only the current selection with a pair of buttons to the right to allow the user to "spin" through the available selections (similar to a slot machine wheel).

If the SPIN control offers the user regularly spaced numeric choices, the RANGE attribute specifies the valid range of values from which the user may choose. The STEP attribute then works in conjunction with RANGE to increment/decrement those values by the specified amount. If the choices are not regular, or are string values, the FROM attribute is used instead of RANGE and STEP. The FROM attribute provides the SPIN control its list of choices from a memory QUEUE or a string. Using the FROM attribute, you may provide the user any type of choices in the SPIN control. The user may select an item from the list or type in the desired value, so this control also acts as an ENTRY control.

Events Generated:

- EVENT:Selected The control has received input focus.
- EVENT:Accepted The user has either selected a value or entered data directly into the control, and moved on to another control.
- EVENT:Rejected The user has entered an invalid value for the entry picture.
- EVENT:NewSelection The user has changed the displayed value.
- EVENT:PreAlertKey The user pressed an ALRT attribute hot key.
- EVENT:AlertKey The user pressed an ALRT attribute hot key.
- EVENT:Drop A successful drag-and-drop to the control.

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
    SPIN(@S8),AT(0,0,20,20),USE(SpinVar1),FROM(Que)
    SPIN(@N3),AT(20,0,20,20),USE(SpinVar2),RANGE(1,999),KEY(F10Key)
    SPIN(@N3),AT(40,0,20,20),USE(SpinVar3),RANGE(5,995),STEP(5)
    SPIN(@S8),AT(60,0,20,20),USE(SpinVar4),FROM(Que),HLP('Check4Help')
    SPIN(@S8),AT(80,0,20,20),USE(SpinVar5),FROM(Que),MSG('Button 3')
    SPIN(@S8),AT(100,0,20,20),USE(SpinVar6),FROM(Que),FONT('Arial',12)
    SPIN(@S8),AT(120,0,20,20),USE(SpinVar7),FROM(Que),DROP
    SPIN(@S8),AT(160,0,20,20),USE(SpinVar8),FROM(Que),IMM
    SPIN(@S8),AT(220,0,20,20),USE(SpinVar9),FROM('Mr|Mrs|Ms'),LEFT
END
```

STRING (declare a string control)

```

STRING(text) ,AT( ) [,CURSOR( )] [,USE( )] [,LAYOUT( )] [,DISABLE] [,FONT( )] [,FULL] [,SCROLL] [,HIDE]
    [,TRN] [,DROPID( )] [,COLOR( )] [,ANGLE( )] [,SKIP
    [, LEFT ]], [, PAGENO ]], [,EXTEND( )]
    [, RIGHT ]
    [, CENTER ] || SUM( ) [, RESET( ) / PAGE ] [, TALLY( ) ] |
    [, DECIMAL ] || AVE( ) [, RESET( ) / PAGE ] [, TALLY( ) ] |
    [, MIN( ) [, RESET( ) / PAGE ] [, TALLY( ) ] |
    [, MAX( ) [, RESET( ) / PAGE ] [, TALLY( ) ] |

```

STRING	Places the <i>text</i> on the WINDOW, TOOLBAR, or REPORT.
<i>text</i>	A string constant containing the text to display, or a display picture token to format the variable specified in the USE attribute (PROP:Text).
AT	Specifies the initial size and location of the control (PROP:AT). If omitted, default values are selected by the runtime library.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the control (PROP:CURSOR). If omitted, the WINDOW's CURSOR attribute is used, else the Windows default cursor is used. Not valid in a REPORT.
USE	A field equate label to reference the control in executable code, or a variable whose contents are displayed in the format of the picture token declared instead of string text (PROP:USE).
LAYOUT	Specifies the control's left-to-right or right-to-left display orientation (PROP:LAYOUT).
DISABLE	Specifies the control appears dimmed when the WINDOW or APPLICATION opens (PROP:DISABLE).
FONT	Specifies the font used to display the text (PROP:FONT).
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter (PROP:FULL). Not valid in a REPORT.
SCROLL	Specifies the control scrolls with the window (PROP:SCROLL). Not valid in a REPORT.
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened (PROP:HIDE). UNHIDE must be used to display it.
TRN	Specifies the text or USE variable characters transparently display over the background (PROP:TRN).
DROPID	Specifies the control may serve as a drop target for drag-and-drop actions (PROP:DROPID). Not valid in a REPORT.

SKIP	Specifies not to print the control if the content is blank, and to move all following controls in the band upward to "fill in" the blank (PROP:SKIP). Valid only in a REPORT.
LEFT	Specifies that the text is left justified within the area specified by the AT attribute (PROP:LEFT).
RIGHT	Specifies that the text is right justified within the area specified by the AT attribute (PROP:RIGHT).
CENTER	Specifies that the text is centered within the area specified by the AT attribute (PROP:CENTER).
DECIMAL	Specifies that the text is aligned on the decimal point within the area specified by the AT attribute (PROP:DECIMAL).
COLOR	Specifies a background color for the control (PROP:COLOR).
ANGLE	Specifies displaying or printing the control at a specified angle measured counter-clockwise from the horizontal or the report's orientation (PROP:ANGLE).
PAGENO	Specifies the current page number is printed in the format of the picture token declared instead of string text (PROP:PAGENO). Valid only in a REPORT.
CNT	Specifies the number of details printed is printed in the format of the picture token declared instead of string text (PROP:CNT). Valid only in a REPORT.
SUM	Specifies the sum of the USE variable is printed in the format of the picture token declared instead of string text (PROP:SUM). Valid only in a REPORT.
AVE	Specifies the average value of the USE variable is printed in the format of the picture token declared instead of string text (PROP:AVE). Valid only in a REPORT.
MIN	Specifies the minimum value of the USE variable is printed in the format of the picture token declared instead of string text (PROP:MIN). Valid only in a REPORT.
MAX	Specifies the maximum value of the USE variable is printed in the format of the picture token declared instead of string text (PROP:MAX). Valid only in a REPORT.
RESET	Specifies the CNT, SUM, AVE, MIN, or MAX is reset when the specified group break occurs (PROP:RESET). Valid only in a REPORT.
PAGE	Specifies the CNT, SUM, AVE, MIN, or MAX is reset to zero when the page break occurs (PROP:PAGE). Valid only in a REPORT.
TALLY	Specifies when to calculate the CNT, SUM, AVE, MIN, or MAX (PROP:TALLY). Valid only in a REPORT.
EXTEND	Specifies special document formatting information. This information contains a valid string of attributes that are assigned to a designated REPORT control for a given document type. Valid only in a REPORT.

The **STRING** control places the *text* on the WINDOW, TOOLBAR, or REPORT at the position and size specified by its AT attribute.

If the *text* parameter is a picture token instead of a string constant, the contents of the variable named in the USE attribute are formatted to that display picture, at the position and size specified by the AT attribute. This makes the STRING with a USE variable a "display-only" control for the variable. The data displayed in the STRING is automatically refreshed every time through the ACCEPT loop, whether the AUTO attribute is present or not.

There is a difference between ampersand (&) use in STRING and PROMPT controls. An ampersand in a STRING displays as part of the *text*, while an ampersand in a PROMPT defines the prompt's "hot" letter.

A STRING with the TRN attribute displays or prints characters transparently, without obliterating the background. This means only the pixels required to create each character are written to screen. This allows the STRING to be placed directly on top of an IMAGE without destroying the background picture.

This control cannot receive input focus.

Events Generated:

EVENT:Drop A successful drag-and-drop to the control.

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
    STRING('String Constant'),AT(10,0,20,20),USE(?S1)
    STRING(@S30),AT(10,20,20,20),USE(StringVar1)
    STRING(@S30),AT(10,20,20,20),USE(StringVar2),CURSOR(CURSOR:Wait)
    STRING(@S30),AT(10,20,20,20),USE(StringVar3),FONT('Arial',12)
END

CustRpt REPORT,AT(1000,1000,6500,9000),THOUS
Break1  BREAK(Pre:Key1)
        HEADER,AT(0,0,6500,1000)
        STRING('Group Head'),AT(3000,500,1500,500),FONT('Arial',18)
        END
Detail  DETAIL,AT(0,0,6500,1000)
        STRING(@N$11.2),AT(6000,1500,500,500),USE(Pre:F1)
        END
        FOOTER,AT(0,0,6500,1000)
        STRING('Group Total:'),AT(5500,500,1500,500)
        STRING(@N$11.2),AT(6000,500,500,500),USE(Pre:F1),SUM,RESET(Pre:Key1)
        END
        END
END
```

TAB (declare a page of a SHEET control)

```
TAB( text ) [,USE()] [,LAYOUT( )] [,KEY()] [,MSG()] [,HLP()] [,REQ] [DROPID()] [,TIP()]
      [,COLOR()] [,FONT()]
```

controls

END

TAB	Declares a group of controls that constitute one of the multiple "pages" of controls contained within a SHEET structure.
<i>text</i>	A string constant containing the text to display on the TAB (PROP:Text).
USE	Specifies a field equate label to reference the control in executable code (PROP:USE).
LAYOUT	Specifies the control's left-to-right or right-to-left display orientation (PROP:LAYOUT)
KEY	Specifies an integer constant or keycode equate that immediately gives focus to the control (PROP:KEY).
MSG	Specifies a string constant containing the default text to display in the status bar when any control in the TAB has focus (PROP:MSG).
HLP	Specifies a string constant containing the default help system identifier for any control in the TAB (PROP:HLP).
REQ	Specifies that when another TAB is selected, the runtime library automatically checks all ENTRY controls in the same TAB structure with the REQ attribute to ensure they contain data other than blanks or zeroes (PROP:REQ).
DROPID	Specifies the control may serve as a drop target for drag-and-drop actions (PROP:DROPID).
TIP	Specifies the text that displays as "balloon help" when the mouse cursor pauses over the control (PROP:ToolTip).
COLOR	Specifies a background color for the control and the default for all controls on the TAB (PROP:COLOR).
FONT	Specifies the font used to display the text on the tab (PROP:FONT). This does not affect the <i>controls</i> placed in the TAB.
<i>controls</i>	Multiple control declarations.

The **TAB** structure declares a group of controls that constitute one of the multiple "pages" of controls contained within a SHEET structure (not valid in a REPORT). The multiple TAB controls in the SHEET structure define the "pages" displayed to the user. The SHEET structure's USE attribute receives the *text* of the TAB control selected by the user.

Input focus changes between the SHEET's TAB controls are signalled only to the SHEET control affected. This means the events generated when the user changes input focus within a SHEET structure are field-specific events for the SHEET control, and the individual TAB controls do not generate events.

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
SHEET,AT(0,0,320,175),USE(SelectedTab)
TAB('Tab One'),USE(?TabOne)
  OPTION('Option 1'),USE(OptVar1),KEY(F10Key),HLP('Option1Help')
  RADIO('Radio 1'),AT(20,0,20,20),USE(?R1)
  RADIO('Radio 2'),AT(40,0,20,20),USE(?R2)
END
OPTION('Option 2'),USE(OptVar2),MSG('Option 2')
RADIO('Radio 3'),AT(60,0,20,20),USE(?R3)
RADIO('Radio 4'),AT(80,0,20,20),USE(?R4)
END
PROMPT('Enter Data:'),AT(100,100,20,20),USE(?P1)
ENTRY(@S8),AT(100,140,32,20),USE(E1)
PROMPT('Enter More Data:'),AT(100,200,20,20),USE(?P2)
ENTRY(@S8),AT(100,240,32,20),USE(E2)
END
TAB('Tab Two'),USE(?TabTwo)
  OPTION('Option 3'),USE(OptVar3)
  RADIO('Radio 1'),AT(20,0,20,20),USE(?R5)
  RADIO('Radio 2'),AT(40,0,20,20),USE(?R6)
END
OPTION('Option 4'),USE(OptVar4)
RADIO('Radio 3'),AT(60,0,20,20),USE(?R7)
RADIO('Radio 4'),AT(80,0,20,20),USE(?R8)
END
PROMPT('Enter Data:'),AT(100,100,20,20),USE(?P3)
ENTRY(@S8),AT(100,140,32,20),USE(E3)
PROMPT('Enter More Data:'),AT(100,200,20,20),USE(?P4)
ENTRY(@S8),AT(100,240,32,20),USE(E4)
END
END
BUTTON('Ok'),AT(100,180,20,20),USE(?Ok)
BUTTON('Cancel'),AT(200,180,20,20),USE(?Cancel)
END
```

See Also:

SHEET

TEXT (declare a multi-line text control)

```
TEXT ,AT( ) [,CURSOR( )][,USE( )][,LAYOUT( )][,DISABLE][,KEY( )][,MSG( )][,HLP( )][,BOXED][,SKIP] [,FONT( )
[,REQ] [,FULL] [,SCROLL] [,ALRT( )] [,HIDE] [,READONLY] [,DROPID( )] [,UPR] [,TRN] [,RTF]
[,TIP( )] [,
| HSCROLL | ] [, | LEFT | ][,COLOR( )][,SINGLE][,RESIZE] [,EXTEND( )]
| VSCROLL | | RIGHT |
| HVSCROLL | | CENTER |
```

TEXT	Places a multi-line data entry field on the WINDOW, TOOLBAR, or REPORT.
AT	Specifies the initial size and location of the control (PROP:AT). If omitted, default values are selected by the runtime library.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the control (PROP:CURSOR). If omitted, the WINDOW's CURSOR attribute is used, else the Windows default cursor is used. Not valid in a REPORT.
USE	The label of the variable that receives the value entered into the control by the user (PROP:USE).
LAYOUT	Specifies the control's left-to-right or right-to-left display and entry orientation (PROP:LAYOUT)
DISABLE	Specifies the control appears dimmed when the WINDOW or APPLICATION opens (PROP:DISABLE).
KEY	Specifies an integer constant or keycode equate that immediately gives focus to the control (PROP:KEY). Not valid in a REPORT.
MSG	Specifies a string constant containing the text to display in the status bar when the control has focus (PROP:MSG). Not valid in a REPORT.
HLP	Specifies a string constant containing the help system identifier for the control (PROP:HLP). Not valid in a REPORT.
BOXED	Specifies a single-track border around a TEXT control, with the text at the top of the border (PROP:BOXED). This attribute only works if the TEXT control is transparent or the parent window does not have the GRAY attribute applied.
SKIP	Specifies the control receives input focus to enter text only with the mouse or accelerator key and does not retain focus (PROP:SKIP). In a REPORT, SKIP specifies not to print the control if the content is blank, and to move all following controls in the band upward to "fill in" the blank.
FONT	Specifies the display font for the control (PROP:FONT).
REQ	Specifies the control may not be left blank or zero (PROP:REQ). Not valid in a REPORT.
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter (PROP:FULL). Not valid in a REPORT.
SCROLL	Specifies the control scrolls with the window (PROP:SCROLL). Not valid in a REPORT.

ALRT	Specifies "hot" keys active for the control (PROP:ALRT). Not valid in a REPORT.
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened (PROP:HIDE). UNHIDE must be used to display it.
READONLY	Specifies the control does not allow data entry (PROP:READONLY). Not valid in a REPORT.
DROPID	Specifies the control may serve as a drop target for drag-and-drop actions (PROP:DROPID). Not valid in a REPORT.
UPR	Specifies all upper case entry (PROP:UPR).
TIP	Specifies the text that displays as "balloon help" when the mouse cursor pauses over the control (PROP:ToolTip). Not valid in a REPORT.
HSCROLL	Specifies that a horizontal scroll bar is automatically added to the text field when any portion of the data lies horizontally outside the visible area (PROP:HSCROLL). Not valid in a REPORT.
VSCROLL	Specifies that a vertical scroll bar is automatically added to the text field when any of the data lies vertically outside the visible area (PROP:VSCROLL). Not valid in a REPORT.
HVSCROLL	Specifies that both vertical and horizontal scroll bars are automatically added to the text field when any portion of the data lies outside the visible area (PROP:HVSCROLL). Not valid in a REPORT.
LEFT	Specifies that the text is left justified within the area specified by the AT attribute (PROP:LEFT).
RIGHT	Specifies that the text is right justified within the area specified by the AT attribute (PROP:RIGHT).
CENTER	Specifies that the text is centered within the area specified by the AT attribute (PROP:CENTER).
COLOR	Specifies a background color for the control (PROP:COLOR).
SINGLE	Specifies the control is only for single line data entry (PROP:SINGLE). This is specifically to allow use of TEXT controls instead of ENTRY for Hebrew or Arabic data entry. Not valid in a REPORT.
RESIZE	Specifies adjusting the print height for the control according to the actual content (PROP:RESIZE). Valid only in a REPORT.
TRN	Specifies the text or USE variable characters transparently display over the background (PROP:Trn).
RTF	Specifies that the contents of the text field supports Rich Text Format.
EXTEND	Specifies special document formatting information. This information contains a valid string of attributes that are assigned to a designated REPORT control for a given document type. Valid only in a REPORT.

The **TEXT** control places a multi-line data entry field on the WINDOW (or TOOLBAR) at the position and size specified by its AT attribute. The variable specified in the USE attribute receives the data entered when the user has completed data entry and moves on to another control. The entered data automatically "word-wraps" to fit in the text box.

The capacity of a TEXT control varies depending on the operating system.

Events Generated:

EVENT:Selected	The control has received input focus.
EVENT:Accepted	The user has completed data entry in the control.
EVENT:PreAlertKey	The user pressed an ALRT attribute hot key.
EVENT:AlertKey	The user pressed an ALRT attribute hot key.
EVENT:Drop	A successful drag-and-drop to the control.

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
    TEXT,AT(0,0,40,40),USE(E1),ALRT(F10Key),CENTER
    TEXT,AT(20,0,40,40),USE(E2),KEY(F10Key),HLP('Text4Help')
    TEXT,AT(40,0,40,40),USE(E3),SCROLL,OVR,UPR
    TEXT,AT(60,0,40,40),USE(E4),CURSOR(CURSOR:Wait),RIGHT
    TEXT,AT(80,0,40,40),USE(E5),DISABLE,FONT('Arial',12)
    TEXT,AT(100,0,40,40),USE(E6),HVSCROLL,LEFT
    TEXT,AT(120,0,40,40),USE(E7),REQ,INS,CAP,MSG('Text Field 7')
END
```

```
CustRpt REPORT,AT(1000,1000,6500,9000),THOUS
Detail  DETAIL,AT(0,0,6500,1000)
    TEXT,AT(0,0,40,40),USE(E1)
    TEXT,AT(100,0,40,40),USE(E6),FONT('Arial',12)
    TEXT,AT(120,0,40,40),USE(E7),CAP
    TEXT,AT(140,0,40,40),USE(E8),UPR
    TEXT,AT(160,0,40,40),USE(E9),LEFT
    TEXT,AT(180,0,40,40),USE(E10),RIGHT
    TEXT,AT(200,0,40,40),USE(E11),CENTER
END
END
```

See Also: ENTRY, PROP:Line , PROP:LineCount

9 - Window and Report Attributes

Attribute Property Equates

Each attribute has a corresponding runtime property listed in its description (PROP:*attribute*). Equates for all runtime properties are contained in the PROPERTY.CLW file. This file also contains equates for the standard values used by some of these properties. Some properties are "read-only" and their value may not be changed, and others are "write-only" properties whose value cannot be determined. These restrictions are noted as applicable.

PROP:Text

PROP:Text is the *text* parameter of an APPLICATION(*text*), WINDOW(*text*), or any control(*text*). This property represents the parameter to any control or window declaration, and could contain any value that is valid as the parameter to the specific control's declaration. For example:

Example:

```
?Image{PROP:Text} = 'My.BMP'           !a new bitmap for the referenced IMAGE control
?Prompt{PROP:Text} = 'New Prompt text' !new text in the referenced PROMPT control
?Entry{PROP:Text} = '@N03'            !new picture for the referenced ENTRY control
```

Attribute Property Parameters

Many attributes take no parameter--they are either present or absent. Therefore, their corresponding runtime properties simply toggle the attribute on or off. Assigning an empty string (") or zero (0) turns them off. Assigning '1' or 1 turns them on. Typically, the standard equates for TRUE and FALSE are used for this purpose. Querying any of these properties returns a blank string when the attribute is not active for the window, report, or control. Examples of these types of attribute properties are: PROP:ABOVE, PROP:ABSOLUTE, and PROP:ALONE.

Example:

```
?MyControl{PROP:DISABLE} = TRUE       !disables the referenced control
```

Many attributes take a single parameter whose presence specifies both the presence of the attribute and its value. Assigning an empty string (") or zero (0) turns them off. Assigning any other valid value turns them on. Examples of these types of attribute properties are PROP:TIMER and PROP:DROP.

Example:

```
MyWindow{PROP:TIMER} = 100           !set the window's timer to 1 second
```

Arrayed Properties

A number of attribute properties are actually arrays which either contain multiple values (such as PROP:ALRT, which may contain up to 255 separately alerted keycodes) or which may be referenced as arrays to directly address their multiple parameters instead of using separately declared equates for each of the individual parameters (like PROP:AT, whose parameters may be addressed either as {PROP:AT,n} or as the separately declared equates for each of the individual parameters: PROP:Xpos, PROP:Ypos, PROP:Width, and PROP:Height).

Example:

```
CheckField STRING(1)
```

```
Screen WINDOW
```

```
    ENTRY(@N3),USE(Ctl:Code)
    ENTRY(@S30),USE(Ctl:Name),REQ
    CHECK('True or False'),USE(CheckField)
    IMAGE('SomePic.BMP'),USE(?Image)
    BUTTON('OK'),USE(?OkButton),KEY(EnterKey)
    BUTTON('Cancel'),USE(?CanxButton),KEY(EscKey)
END
```

```
CODE
```

```
OPEN(Screen)
```

```
Screen{PROP:AT,1} = 0           !Position window to top left corner
Screen{PROP:AT,2} = 0
Screen{PROP:GRAY} = 1         !Give window 3D lock
Screen{PROP:STATUS,1} = -1    !Create status bar with two sections
Screen{PROP:STATUS,2} = 180
Screen{PROP:STATUS,3} = 0     !Terminate status bar array
Screen{PROP:StatusText,2} = FORMAT(TODAY(),@D2) !Put date in status bar section 2:
?CtlCode{PROP:ALRT,1} = F10Key !Alert F10 on Ctl:Code entry control
?CtlCode{PROP:Text} = '@N4'   !Change entry picture token
?Image{PROP:Text} = 'MyPic.BMP' !Change image control filename
?OkButton{PROP:DEFAULT} = '1' !Put DEFAULT attribute on OK button

?MyButton{PROP:ICON} = 'C:\Windows\MORICONS.DLL[10]'
!Display 11th icon in MORICONS.DLL (zero-based)

?MyButton{PROP:ICON} = 'C:\Windows\MORICONS.DLL[0]'
!Display first icon in MORICONS.DLL (zero-based)

?CheckField{PROP:TrueValue} = 'T'           !Checked/unchecked values for CHECK co
?CheckField{PROP:FalseValue} = 'F'
ACCEPT
END
```

Window and Report Attributes

ABSOLUTE (set fixed-position printing)

ABSOLUTE

The **ABSOLUTE** attribute (PROP:ABSOLUTE) ensures that the DETAIL, or group HEADER or FOOTER structure (contained within a BREAK structure), always prints at a fixed position on the page. When ABSOLUTE is present, the position specified by the x and y parameters of the structure's AT attribute is relative to the top left corner of the page. ABSOLUTE has no effect on following structures printed without the ABSOLUTE attribute.

Example:

```
CustRpt REPORT,AT(1000,2000,6500,9000),THOUS
      HEADER
      !structure elements
      END
CDetail1 DETAIL,AT(0,0,6500,1000)
      !structure elements
      END
CDetail2 DETAIL,AT(1000,1000,6500,1000),ABSOLUTE !fixed position detail
      !structure elements
      END
      END
```

ALONE (set to print without page header, footer, or form)**ALONE**

The **ALONE** attribute (PROP:ALONE) specifies that the DETAIL, or group HEADER or FOOTER structure (contained within a BREAK structure), is to print on the page without any FORM, or page HEADER or FOOTER (not within a BREAK structure). The normal use is for report title and grand total pages.

Example:

```
CustRpt  REPORT
TitlePage  DETAIL,ALONE      !Title page detail structure
          !structure elements
          END
CustDetail DETAIL
          !structure elements
          END
          FOOTER
          !structure elements
          END
          END
```

ALRT (set window "hot" keys)

ALRT(*keycode*)

ALRT Specifies a "hot" key active while the APPLICATION, WINDOW, or control on which it is placed has focus.

keycode A numeric constant keycode or keycode EQUATE.

The **ALRT** attribute (PROP:ALRT) specifies a "hot" key active while the APPLICATION, WINDOW, or control on which it is placed has focus.

When the user presses an ALRT "hot" key, two events (field-independent if the ALRT is on an APPLICATION or WINDOW, field-specific if the ALRT is on a control), EVENT:PreAlertKey and EVENT:AlertKey, are generated (in that order). If the code does not execute a CYCLE statement when processing EVENT:PreAlertKey, you "shortstop" the library's default action on the alerted keypress. If the code does execute a CYCLE statement when processing EVENT:PreAlertKey, the library performs its default action for the alerted keypress. In either case, EVENT:AlertKey is generated following EVENT:PreAlertKey. When EVENT:AlertKey is generated, the USE variable of the control with input focus is not automatically updated (use UPDATE if this is required).

You may have multiple ALRT attributes on one APPLICATION, WINDOW, or control (up to 255). The ALERT statement and the ALRT attribute of a window or control are completely separate. This means that clearing ALERT keys has no effect on any keys alerted by ALRT attributes.

PROP:ALRT is an array, containing up to 255 keycodes. The array element number actually used is internally assigned to the first free array element if the specified element number is larger than the current number of assigned keycodes. For example, assuming there are no keys alerted at all, if you specify assigning to element number 255, it is actually assigned to element number 1. Subsequently assigning another keycode to element number 255 (still free), it is actually assigned to element number 2. Explicitly assigning a keycode to element number 1, however, overwrites any other keycode already assigned to element number 1.

Example:

```
WinOne WINDOW,AT(0,0,160,400)
    ENTRY,AT(6,40),USE(SomeVar1),ALRT(MouseLeft) !Mouse click alerted for control
    ENTRY,AT(60,40),USE(SomeVar2),ALRT(F10Key) !F10 alerted for control
    END
CODE
OPEN(WinOne)
ACCEPT
CASE FIELD()
OF ?SomeVar1
CASE EVENT()
OF EVENT:PreAlertKey !Pre-check alert events
CYCLE !Allow standard NouseLeft action to process
OF EVENT:AlertKey !Alert processing
DO ClickRoutine
END
OF ?SomeVar2
CASE EVENT()
OF EVENT:AlertKey !Alert processing
DO F10Routine
END
END
END
```

ANGLE (set control display or print angle)**ANGLE**(*size*)

ANGLE	Defines the orientation of a STRING control.
<i>size</i>	An integer constant or constant expression that specifies the amount of rotation, in tenths of degrees. If positive, the angle is measured counter-clockwise from the report's horizontal orientation. Valid values are between 3600 and -3600.

The **ANGLE** attribute (PROP:ANGLE) specifies displaying or printing the STRING control at a specified angle measured counter-clockwise from the horizontal of the window or horizontal orientation of the report (either Portrait or Landscape). This allows you to display or print text at any angle in addition to the standard horizontal. The FONT for the STRING control must be a TrueType font.

Example:

```
WinOne WINDOW,AT(0,0,160,400),FONT('Arial')
    !Display Horizontal text
    STRING('String Constant'),AT(6,40),USE(?String1)
    !Display Vertical text
    STRING('String Constant'),AT(6,40),USE(?String2),ANGLE(900)
    !Display upside-down text
    STRING('String Constant'),AT(6,40),USE(?String3),ANGLE(1800)
END

CustRpt REPORT,AT(1000,1000,6500,9000),THOUS,FONT('Arial',10)
Detail  DETAIL,AT(0,0,6500,1000)
    !Print Horizontal text
    STRING('String Constant'),AT(500,500,1500,500)
    !Print Vertical text
    STRING('String Constant'),AT(500,500,1500,500),ANGLE(900)
    !Print upside-down text
    STRING('String Constant'),AT(500,500,1500,500),ANGLE(1800)
    END
END
```

AT (set position and size)

AT([*x*] [,*y*] [,*width*] [,*height*])

AT	Defines the position and size of the structure or control on which it is placed.
<i>x</i>	An integer constant or constant expression that specifies the horizontal position of the top left corner (PROP:Xpos, equivalent to {PROP:At,1}). If omitted, the runtime library provides a default value.
<i>y</i>	An integer constant or constant expression that specifies the vertical position of the top left corner (PROP:Ypos, equivalent to {PROP:At,2}). If omitted, the runtime library provides a default value.
<i>width</i>	An integer constant or constant expression that specifies the width (PROP:Width, equivalent to {PROP:At,3}). If omitted, the runtime library provides a default value.
<i>height</i>	An integer constant or constant expression that specifies the height (PROP:Height, equivalent to {PROP:At,4}). If omitted, the runtime library provides a default value.

The **AT** attribute (PROP:AT) defines the position and size of the structure or control on which it is placed. The *x,y* position is relative and dependent upon the statement on which the AT attribute is placed.

The values contained in the *x*, *y*, *width*, and *height* parameters are measured in dialog units for an APPLICATION or WINDOW. The *x*, *y*, *width*, and *height* parameters on a REPORT without the THOUS, MM, or POINTS attribute are also measured in dialog units.

Dialog units are defined as one-quarter the average character width by one-eighth the average character height. The actual size of a dialog unit is dependent upon the size of the default font for the window or report. This measurement is based on the font specified in the FONT attribute of the window or report, or the system default font specified by Windows (if there is no FONT attribute on the window or report).

Window Usage

The *x* and *y* parameters are relative to the top left corner of the video screen when the AT attribute is on an APPLICATION structure, or a WINDOW without the MDI attribute that is opened before an APPLICATION structure is opened by the program.

The *x* and *y* parameters are relative to the top left corner of the APPLICATION's client area when the AT attribute is placed on a WINDOW with the MDI attribute, or a WINDOW without the MDI attribute opened after an APPLICATION structure has been opened.

The *width* and *height* parameters specify the size of the "client area" or "workspace" of an APPLICATION. This is the area below the MENUBAR and above the status bar which defines the area in which the TOOLBAR is placed and MDI "child" windows are opened. On a WINDOW, they specify the size of the "workspace" which may contain control fields.

Window Control Usage

The *x* and *y* parameters are relative to the top left corner of the APPLICATION or WINDOW's client area.

REPORT Structure Usage

The AT attribute on a REPORT structure defines the position and size of the area of the page devoted to printing report detail. This is the area in which all DETAIL structures and any group HEADER and FOOTER structures contained within BREAK structures will print.

Print Structure Usage

The AT attribute on print structures performs two different functions, depending upon the structure on which it is placed.

When placed on a FORM, or page HEADER or FOOTER (not within a BREAK structure), the AT attribute defines the position and size on the page at which the structure prints. The position specified by the *x* and *y* parameters is relative to the top left corner of the page.

When placed on a DETAIL, or group HEADER or FOOTER (contained within a BREAK structure), the print structure prints according to the following rules (unless the ABSOLUTE attribute is also present):

- The *width* and *height* parameters of the AT attribute specify the minimum print size of the structure.
- The structure actually prints at the next available position within the detail print area (specified by the REPORT's AT attribute).
- The position specified by the *x* and *y* parameters of the structure's AT attribute is an offset from the next available print position within the detail print area.
- The first print structure on the page prints at the top left corner of the detail print area (at the offset specified by its AT attribute).
- Next and subsequent print structures print relative to the ending position of the previous print structure:
 - If there is room to print the next structure beside the previous structure, it prints there.
 - If not, it prints below the previous.

REPORT Control Usage

The x and y parameters are relative to the top left corner of the print structure containing the control.

Example:

```

WinOne WINDOW,AT(0,0,380,200),MDI !top left corner, relative to app frame
      END
WinTwo WINDOW,AT(0,0,380,200)      !Top left corner, relative to video screen
      END
!Measurement in dialog units
WinOne WINDOW,AT(0,0,160,400)
      ENTRY,AT(8,40,80,8)          !Approx. 2 characters in, 5 down, 20 wide, 1 high
      END

CustRpt REPORT,AT(1000,1000,6500,9000),THOUS !AT specifies detail print area
Detail  DETAIL,AT(0,0,6500,1000)             !AT specifies band size and
                                             !relative position offset from
                                             !last printed detail
      STRING('String Constant'),AT(500,500,1500,500)
                                             !AT specifies control size and
                                             !offset within the detail band
      END
      END

CustRpt REPORT,AT(1000,2000,6500,7000),THOUS !1" margins all around
      HEADER,AT(1000,1000,6500,1000)         !Page relative position
      !structure elements                     !1" band across top of page
      END
CustD1  DETAIL,AT(0,0,6500,1000)             !Detail relative position
      !structure elements                     !1" band across page
      END
CustD2  DETAIL,ABSOLUTE,AT(1000,8000,6500,1000) !Page relative position
      !structure elements                     !1" band near page bottom
      END
      FOOTER,AT(1000,9000,6500,1000)         !Page relative position
      !structure elements                     !1" band across page bottom
      END
      END
CustRpt1 REPORT,AT(1000,1000,6500,9000),THOUS !1" margins all around for
                                             ! detail area on 8.5" x 11"
      !report declarations
      END
CustRpt2 REPORT,AT(72,72,468,648),POINTS    !1" margins all around for
                                             !detail area on 8.5" x 11"
      !report declarations
      END

```

See Also: SETPOSITION, GETPOSITION

AUTO (set USE variable automatic re-display)**AUTO**

The **AUTO** attribute (PROP:AUTO) specifies all window and toolbar controls' USE variables re-display on screen each time through the ACCEPT loop. This incurs some overhead, but ensures the data displayed is current, without requiring explicit DISPLAY statements.

Example:

```
WinOne WINDOW,AT(,,380,200),MDI,CENTER,AUTO !All controls values always display
      !controls
      END
CODE
!ACCEPT automatically re-displays changed USE variables
ACCEPT
END
```

AUTOSIZE (set OLE object resizing)**AUTOSIZE**

The **AUTOSIZE** attribute (PROP:AUTOSIZE, write-only) specifies the OLE object automatically resizes itself when the OLE container control's AT attribute parameters change at runtime using property syntax to change the values of PROP:AT.

AVE (set report total average)

AVE([*variable*])

AVE	Calculates the average (arithmetic mean) of the STRING controls' USE variable is printed.
<i>variable</i>	The label of a numeric variable to receive the intermediate values calculated for the AVE. This allows you to create totals on other totals. The value in the <i>variable</i> is internally updated by the print engine, so it is only useful for use within the REPORT structure.

The **AVE** attribute (PROP:AVE) specifies printing the average (arithmetic mean) of the STRING controls' USE variable. Unless the TALLY attribute is present, the result is calculated as follows:

- An AVE field in a DETAIL structure is calculated each time the DETAIL structure containing the control PRINTs.
- An AVE field in a group FOOTER structure is calculated each time any DETAIL structure in the BREAK structure containing the control PRINTs.
- An AVE field in a page FOOTER structure is calculated each time any DETAIL structure in any BREAK structure PRINTs.
- An AVE field in a HEADER is meaningless, since no DETAIL structures will have been printed at the time the HEADER is printed.

The average is reset only if the RESET or PAGE attribute is also specified. The STRING control using this attribute would usually be placed in a group or page FOOTER.

Example:

```
CustRpt REPORT,AT(1000,1000,6500,9000),THOUS
Break1 BREAK(LocalVar),USE(?BreakOne)
Break2 BREAK(Pre:Key1),USE(?BreakTwo)
Detail DETAIL,AT(0,0,6500,1000),USE(?DetailOne)
        STRING(@N$11.2),AT(6000,1500,500,500),USE(Pre:F1)
        END
        FOOTER,AT(0,0,6500,1000),USE(?BreakOneGroupFoot)
        STRING('Group Average: '),AT(5500,500)
        STRING(@N$11.2),AT(6000,500),USE(Pre:F1),AVE(LocalVar),RESET(Break2)
        END
        END
        FOOTER,AT(0,0,6500,1000),USE(?BreakOneGroupFoot)
        STRING('Grand Average: '),AT(5500,500)
        STRING(@N$11.2),AT(6000,500),USE(LocalVar),AVE,TALLY(?BreakTwo)
        END
        END
        END
```

BEVEL (set 3-D effect border)**BEVEL**(*outer* [,*inner*] [,*style*])

BEVEL	Specifies a 3-D effect border on a control.
<i>outer</i>	An integer constant or constant expression that specifies the width of the outer edge of the bevel (PROP:BevelOuter, equivalent to {PROP:Bevel,1}). If negative, the outer edge appears to be lowered; if positive, the outer edge appears to be raised.
<i>inner</i>	An integer constant or constant expression that specifies the width of the inner edge of the bevel (PROP:BevelInner, equivalent to {PROP:Bevel,2}). If negative, the inner edge appears to be lowered; if positive, the inner edge appears to be raised. If omitted, there is no inner edge to the border.
<i>style</i>	An integer constant or constant expression that specifies fine control of the bevel, overriding the signs of the <i>outer</i> and <i>inner</i> parameters (PROP:BevelStyle, equivalent to {PROP:Bevel,3}).

The **BEVEL** attribute (PROP:BEVEL) of a PANEL, OPTION, GROUP, or REGION control specifies a 3-D effect border. The signs of the *outer* and *inner* parameters determine whether the control appears to be raised or lowered. The *style* parameter allows fine control of the bevel. This parameter is a bitmap with the bits apportioned for each edge as follows:

```
Bits:   15 - 12 | 11 - 08 | 07 - 04 | 03 - 00
Edge:   left  | top    | right | bottom
```

Each of these four-bit nibbles is further divided into two two-bit sections that govern the appearance of the inner and outer parts of the edge. The low order two bits of each nibble defines the outer part, while the high order bits define the inner:

```
Binary:  00b   | 01b   | 10b   | 11b
Result: no edge | raised | lowered | gray
```

Combining these into nibbles creates the definition of one edge of the bevel:

```
0110b = raised inner, lowered outer
1001b = lowered inner, raised outer
```

Example:

```
Win1 WINDOW,AT(0,0,160,400)
  PANEL,AT(25,15,50,50),USE(?Panel1),BEVEL(5,-5) !Raised outer, lowered inner
  PANEL,AT(0,0,,),USE(?Panel2),FULL,BEVEL(2,2,1111010110101001b)
    !left  = all gray
    !top   = inner raised, outer raised
    !right = inner lowered, outer lowered
    !bottom = inner lowered, outer raised
  REGION,AT(0,80,5,,),USE(?ResizeBar),FULL,IMM,BEVEL(2,2,0101000010100000b)
    !A vertical resize bar
END
```

BOXED (set controls group border)**BOXED**

The **BOXED** attribute (PROP:BOXED) specifies a single-track border around a TEXT control, or a GROUP or OPTION structure. The *text* parameter appears in a gap at the top of the border box. If BOXED is omitted, the *text* parameter is not printed or displayed on screen.

CAP, UPR (set case)**CAP****UPR**

The **CAP** and **UPR** attributes specify the automatic case of text entered into ENTRY or TEXT controls when the MASK attribute is on the window or of text printed in a TEXT control.

The UPR attribute (PROP:UPR) specifies all upper case.

The CAP attribute (PROP:CAP) specifies "Proper Name Capitalization," where the first letter of each word is capitalized and all other letters are lower case. The user can override this default behavior by pressing the SHIFT key to allow an upper case letter in the middle of a name (allowing for names such as, "McDowell") or SHIFT while CAPS-LOCK is on, forcing a lower case first letter (allowing for names such as, "von Richtofen").

CENTER (set centered window position)**CENTER**

The **CENTER** attribute (PROP:CENTER) indicates that the window's default position is centered. A WINDOW structure with the MDI attribute is centered on the APPLICATION. An APPLICATION structure is centered on the screen. A non-MDI WINDOW is centered on its parent (the window currently with focus when the non-MDI WINDOW is opened).

This attribute has no meaning unless at least one parameter of the AT attribute is omitted. This means that the CENTER attribute provides a default value for any omitted AT parameter.

Example:

```
WinOne WINDOW,AT(,,380,200),MDI,CENTER !Window centered relative to application frame
      END
```

```
WinTwo WINDOW,AT(,,380,200),CENTER      !Window centered relative to its parent:
      END
```

CENTERED (set centered image)**CENTERED**

The **CENTERED** attribute (PROP:CENTERED) indicates an image displayed at its default size and centered in its display area:

- On an IMAGE control, the image is centered in the area specified by the AT attribute.
- On a TOOLBAR with the WALLPAPER attribute, the toolbar's background image is centered in the toolbar.
- On an APPLICATION or WINDOW with the WALLPAPER attribute, the window's background image is centered in the client area of the window.

Example:

```
MDIChild WINDOW('Child One'),MDI,SYSTEM,MAX
  MENUBAR
    MENU('Edit'),USE(?EditMenu)
    ITEM('Undo'),USE(?UndoText),KEY(CtrlZ),STD(STD:Undo)
    ITEM('Cu&t'),USE(?CutText),KEY(CtrlX),STD(STD:Cut)
    ITEM('Copy'),USE(?CopyText),KEY(CtrlC),STD(STD:Copy)
    ITEM('Paste'),USE(?PasteText),KEY(CtrlV),STD(STD:Paste)
  END
END
TOOLBAR,USE(?Toolbar),WALLPAPER('MyWall.GIF'),CENTERED
  BUTTON('Cut'),USE(?CutButton),ICON(ICON:Cut),STD(STD:Cut),FLAT
  BUTTON('Copy'),USE(?CopyButton),ICON(ICON:Copy),STD(STD:Copy),FLAT
  BUTTON('Paste'),USE(?PasteButton),ICON(ICON:Paste),STD(STD:Paste),FLAT
END
END

WinOne WINDOW,AT(,,380,200),MDI,WALLPAPER('MyWall.GIF'),CENTERED
  END
WinOne WINDOW,AT(,,380,200),MDI
  IMAGE('MyWall.GIF'),AT(0,0,380,200),CENTERED
  END
```

See Also:

WALLPAPER

TILED

CHECK (set on/off ITEM)**CHECK**

The **CHECK** attribute (PROP:CHECK) specifies an ITEM that may be either ON or OFF. When ON, a check appears to the left of the menu selection and the USE variable receives the value one (1). When OFF, the check to the left of the menu selection disappears and the USE variable receives the value zero (0).

CLIP (set OLE object clipping)**CLIP**

The **CLIP** attribute (PROP:CLIP, write-only) specifies the OLE object only displays what fits into the size of the OLE container control's AT attribute. If the object is larger than the OLE container control, only the top left corner displays.

CNT (set total count)

CNT([*variable*])

CNT	Calculates the number of times DETAIL structures have been printed.
<i>variable</i>	The label of a numeric variable to receive the intermediate values calculated for the CNT. This allows you to create totals on other totals. The value in the <i>variable</i> is internally updated by the print engine, so it is only useful for use within the REPORT structure.

The **CNT** attribute (PROP:CNT) specifies an automatic count of the number of times DETAIL structures have been printed. Unless the TALLY attribute is present, the result is calculated as follows:

- A CNT field in a DETAIL structure increments each time the DETAIL structure containing the control PRINTs. This provides a "running" count.
- A CNT field in a group FOOTER structure increments each time any DETAIL structure in the BREAK structure containing the control PRINTs. This provides a total of the number of DETAIL structures printed in the group.
- A CNT field in a page FOOTER structure increments each time any DETAIL structure in any BREAK structure PRINTs. This provides a total of the number of DETAIL structures printed on the page (or report).
- A CNT field in a HEADER is meaningless, since no DETAIL structures will have been printed at the time the HEADER is printed.

The CNT is reset only if the RESET or PAGE attribute is also specified.

Example:

```

CustRpt REPORT,AT(1000,1000,6500,9000),THOUS
Break1  BREAK(LocalVar),USE(?BreakOne)
Break2  BREAK(Pre:Key1),USE(?BreakTwo)
Detail  DETAIL,AT(0,0,6500,1000),USE(?DetailOne)
        STRING(@N$11.2),AT(6000,1500,500,500),USE(Pre:F1)
        END
        FOOTER,AT(0,0,6500,1000),USE(?BreakOneGroupFoot)
        STRING('Group Count:'),AT(5500,500)
        STRING(@N$11.2),AT(6000,500),USE(Pre:F1),CNT(LocalVar),RESET(Break2)
        END
        END
        FOOTER,AT(0,0,6500,1000),USE(?BreakOneGroupFoot)
        STRING('Grand Count:'),AT(5500,500)
        STRING(@N$11.2),AT(6000,500),USE(LocalVar),CNT,TALLY(?BreakTwo)
        END
        END
        END

```

COLOR (set color)

COLOR(*color* [, *selected fore*] [, *selected back*])

COLOR	Specifies display or print color.
<i>color</i>	Specifies the background color (PROP:Background or PROP:FillColor, equivalent to {PROP:Color,1}). Foreground color is specified in the FONT attribute.
<i>selected fore</i>	Specifies the default foreground color for the selected text on a control that can receive focus (PROP:SelectedColor, equivalent to {PROP:Color,2}). Not valid in a REPORT.
<i>selected back</i>	Specifies the default background color for the selected text on a control that can receive focus (PROP:SelectedFillColor, equivalent to {PROP:Color,3}). Not valid in a REPORT.

The **COLOR** attribute (PROP:COLOR) specifies the default background and selected foreground and background colors.

The color values in each of the three parameters are constants which contain the red, green, and blue components to create the color in the three low-order bytes of a LONG value (bytes 0, 1, and 2: Red = 000000FFh, Green = 0000FF00h, and Blue = 00FF0000h), or EQUATEs for a standard Windows color value (which are all negative values). EQUATEs for Windows' standard colors are contained in the EQUATES.CLW file. Each of the runtime properties returns COLOR:None if the associated parameter is absent.

Windows automatically finds the closest match to the specified color value for the hardware on which the program is run. Windows standard colors may be reconfigured by the user in the Windows Control Panel. Any control using a Windows standard color is automatically repainted with the new color when this occurs.

WINDOW and TOOLBAR Usage

On a WINDOW or TOOLBAR, the COLOR attribute specifies the background display color of the WINDOW or TOOLBAR and the default background and selected foreground and background colors for all controls in the WINDOW or TOOLBAR without their own COLOR attribute.

Window Control Usage

The COLOR attribute specifies the display color of a LINE control. On a BOX, ELLIPSE, or REGION control, the *color* parameter specifies the color used for the control's border. On all other controls, the *color* parameter specifies the background control color, overriding the user's standard Windows color scheme for that control type.

For most of those controls that can receive focus, the *selected fore* and *selected back* parameters specify the foreground and background colors of the selected text or item.

Report Usage

On a REPORT statement, the COLOR attribute specifies the background print color of the REPORT and the default background color for all DETAIL, HEADER, FOOTER, or FORM in the REPORT without a COLOR attribute.

The COLOR attribute specifies the background print color of the DETAIL, HEADER, FOOTER, or FORM on which it is placed, and the default background color for all controls in the DETAIL, HEADER, FOOTER, or FORM without a COLOR attribute.

The COLOR attribute specifies the print color of a LINE control, specifies the border color of a BOX or ELLIPSE control, or the background color of any other control.

Example:

```
WinOne WINDOW,AT(0,0,160,400),COLOR(00FF0000h,0000FF00h,000000FFh)
      !Blue background, Green selected foreground, Red selected background
      TOOLBAR,COLOR(00FF0000h,0000FF00h,000000FFh)
      !Blue background, Green selected foreground, Red selected background
      BOX,AT(20,20,20,20),COLOR(COLOR:ACTIVEBORDER) !Windows' active border color
      END
      BOX,AT(100,100,20,20),COLOR(00FF0000h)           !Blue
      BOX,AT(140,140,20,20),COLOR(0000FF00h)         !Green
      BOX,AT(180,180,20,20),COLOR(000000FFh)         !Red
      END

CustRpt  REPORT,AT(1000,1000,6500,9000),THOUS,COLOR(00FF0000h)  !Blue background
CustDetail  DETAIL,AT(0,0,6500,1000)
      ELLIPSE,AT(60,60,200,200),COLOR(COLOR:ACTIVEBORDER)  !Color EQUATE
      BOX,AT(360,60,200,200),COLOR(00FF0000h)              !Pure Red
      END
      END

RptOne  REPORT,AT(0,0,160,400),COLOR(00FF0000h)           !Blue default background
      HEADER,COLOR(0000FF00h)                               !Green page header background
      !structure elements
      END

CustDl  DETAIL                                           !uses the default background color
      !structure elements
      END
      FOOTER,COLOR(000000FFh)                             !Red page footer background
      !structure elements
      END
      END
```

COLUMN (set list box highlight bar)**COLUMN**

The **COLUMN** attribute (PROP:COLUMN) specifies a field-by-field highlight bar on a LIST or COMBO control with multiple display columns. PROP:COLUMN returns zero (0) if off, else it returns the currently highlighted column number.

EXTEND (set document formatting)

EXTEND([*attributelist* [,*attributelist*...])

EXTEND	Specifies the attributes needed for a particular document type.
<i>attributelist</i>	A string constant, variable, or EQUATE containing an optional type of output document and its associated attributes as described below:
<i>type</i>	A string constant that specifies which output format receives the attribute list
<i>attribute</i>	A string constant containing the valid formatting required for the designated document type.

The **EXTEND** attribute (PROP:EXTEND) specifies a valid string of attributes that are assigned to a designated REPORT control for a given document *type*. Current possible values for *type* are HTML, XML, TXT, PDF, or ALL.

The contents of *attribute* are dependant on the target type. If a vertical bar (|) or parentheses (")) is needed to appear in the attribute, then the attribute needs to be surrounded by single quotes (').

Each *attributelist* may designate an optional document *type*, and associated attributes for that document type, using the following syntax:

[*type*](*attribute* [*attribute*...])

Example:

```

IF pAttribute = TargetAttr:HIDE AND pReportGeneratorType=RepGen:ALL THEN
  SELF.R $ pReportControl{PROP:width}=0
  SELF.R $ pReportControl{PROP:height}=0
  RETURN
END
lAttribute = SELF.R$pReportControl{PROP:Extend}

```

COMPATIBILITY (set OLE control compatibility)

COMPATIBILITY(*mode*)

COMPATIBILITY

Specifies OLE control compatibility setting.

mode An integer constant for the compatibility setting.

The **COMPATIBILITY** attribute (PROP:COMPATIBILITY, write-only) specifies a compatibility mode for certain OLE or .OCX objects that require it. The *mode* should in general be zero (0), however some OLE objects (like Windows bitmap editor) do not work unless it is set to one (1).

Example:

```
WinOne WINDOW,AT(0,0,200,200)
        OLE,AT(10,10,160,100),USE(?OLEObject),CREATE('Excel.Sheet.5'),COMPATIBILITY(0)
        END
END
```

CREATE (create OLE control object)

CREATE(*server* [, *object*])

CREATE	Specifies creating a new object for the OLE control.
<i>server</i>	A string constant containing the name of an OLE Server application, as it appears in the operating system's registry.
<i>object</i>	A string constant containing the name of the OLE Compound Storage file and the object within it to open.

The **CREATE** attribute (PROP:CREATE, write-only) specifies the OLE control creates a new OLE or .OCX object. The *server* value is the object name as it appears in the Operating System's Registry Settings (in Win95, this information is available in REGEDIT.EXE under HKEY_CLASSES_ROOT, or in the Microsoft System Information program that comes with Microsoft Office--MSINFO32.EXE).

When the *object* parameter is present, CREATE operates just as the OPEN attribute does, opening the saved *object* for the OLE control from an OLE Compound Storage file (and ignoring the *server* parameter). When the object is opened, the saved version of the container properties are re-loaded, so properties do not need to be specified on an object opened. The *object* parameter syntax must take the form: *Filename!ObjectName*.

Example:

```
WinOne WINDOW,AT(0,0,200,200)
    OLE,AT(10,10,160,100),USE(?OLEObject),CREATE('Excel.Sheet.5')
    END
END
```

CURSOR (set mouse cursor type)**CURSOR**(*file*)

CURSOR	Specifies a mouse cursor to display.
<i>file</i>	A string constant containing the name of a .CUR file, or an EQUATE naming a Windows-standard mouse cursor. The .CUR file is linked into the .EXE as a resource.

The **CURSOR** attribute (PROP:CURSOR) specifies a mouse cursor to be displayed when the mouse is positioned over the APPLICATION, WINDOW, TOOLBAR, or control. This cursor is inherited by the controls in the APPLICATION, WINDOW, or TOOLBAR unless overridden. Windows 3.1 only supports monochrome cursors (326-byte .CUR files).

EQUATE statements for the Windows-standard mouse cursors are contained in the EQUATES.CLW file. The following list is a representative sample of these (see EQUATES.CLW for the complete list):

CURSOR:None	No mouse cursor
CURSOR:Arrow	Normal windows arrow cursor
CURSOR:IBeam	Capital "I" like a steel I-beam
CURSOR:Wait	Hourglass
CURSOR:Cross	Large plus sign
CURSOR:UpArrow	Vertical arrow
CURSOR:Size	Four-headed arrow
CURSOR:Icon	Box within a box
CURSOR:SizeNWSE	Double-headed arrow slanting left
CURSOR:SizeNESW	Double-headed arrow slanting right
CURSOR:SizeWE	Double-headed horizontal arrow
CURSOR:SizeNS	Double-headed vertical arrow
CURSOR:DragWE	Double-headed horizontal arrow

Example:

```

!Window with custom cursor
WinTwo WINDOW,CURSOR('CUSTOM.CUR')
    TOOLBAR,CURSOR('CURSOR:Cross')           !Toolbar with large plus sign cursor
    BUTTON('Cut'),USE(?CutButton),STD(STD:Cut)
    BUTTON('Copy'),USE(?CopyButton),STD(STD:Copy)
    BUTTON('Paste'),USE(?PasteButton),STD(STD:Paste)
    END
    REGION,AT(20,20,20,20),CURSOR(CURSOR:IBeam) !Region with I-beam cursor
    REGION,AT(100,100,20,20)
    END

```

DEFAULT (set enter key button)**DEFAULT**

The **DEFAULT** attribute (PROP:DEFAULT) specifies a **BUTTON** that is automatically pressed when the user presses the **ENTER** key. Only one active **BUTTON** on a window should have this attribute.

DELAY (set repeat button delay)**DELAY**(*time*)**DELAY** Specifies the delay between first and second event generation.*time* An integer constant containing the time delay to set, in hundredths of a second.

The **DELAY** attribute (PROP:DELAY) specifies the delay between first and second event generation for automatically repeating buttons. For a **BUTTON** control with the **IMM** attribute, this is the time between the first and second **EVENT:Accepted**. For a **SPIN** control, this is the time between the first and second **EVENT:NewSelection** generated by the spin buttons.

The purpose of the **DELAY** attribute is to change the delay time from its default value so that users do not inadvertently begin repeating the action when that is not their intention. Assigning a zero to **PROP:DELAY** resets the default setting, any other value sets the repeat delay for the control.

Example:

```
MDIChild WINDOW( 'Child One' ), AT( 0, 0, 320, 200 ), MDI, MAX, HVSCROLL
    BUTTON( 'Press Me' ), AT( 10, 10, 40, 20 ), USE( ?PressMe ), IMM, DELAY( 100 ) !1 second
    SPIN( @n3 ), AT( 60, 10, 40, 10 ), USE( SpinVar ), RANGE( 0, 999 ), DELAY( 100 ) !1 second
END
```

CODE

OPEN(MDIChild)

?PressMe{PROP:Delay} = 5 0 !Reset delay to 1/2 second

?SpinVar{PROP:Delay} = 5 0 !Reset delay to 1/2 second

?PressMe{PROP:Repeat} = 5 !Set repeat to 5 hundredths of a second

?SpinVar{PROP:Repeat} = 5 !Set repeat to 5 hundredths of a second

See Also:

IMM

REPEAT

DISABLE (set control dimmed at open)**DISABLE**

The **DISABLE** attribute (PROP:Disable) specifies a control that is disabled when the WINDOW or APPLICATION is opened. The disabled control may be activated with the ENABLE statement. PROP:Disable returns a (1) if the control is DISABLEd and a (0) if ENABLEd.

DOCK (set dockable toolbox window)**DOCK**(*positions*)**DOCK** Specifies a dockable TOOLBOX.*positions* A bitmap specifying the edges available for docking.

The **DOCK** attribute (PROP:DOCK) specifies a WINDOW with the TOOLBOX attribute which may be docked to an edge of the application frame. The following EQUATEs for standard *positions* values are contained in EQUATES.CLW:

DOCK:Left	EQUATE(1)
DOCK:Top	EQUATE(2)
DOCK:Right	EQUATE(4)
DOCK:Bottom	EQUATE(8)
DOCK:Float	EQUATE(16)
DOCK:All	EQUATE(31)

Example:

```
Win1 WINDOW('Tools'),TOOLBOX,DOCK(DOCK:Left+DOCK:Right) !Dockable left and right only
      BUTTON('Date'),USE(?Button1)
      BUTTON('Time'),USE(?Button2)
      END
```

See Also:

DOCKED

TOOLBOX

DOCKED (set dockable toolbox window docked at open)**DOCKED**(*position*)**DOCKED** Specifies a dockable TOOLBOX docked at open.*position* A bitmap specifying the edge to which it is docked.

The **DOCKED** attribute (PROP:DOCKED) specifies a WINDOW with the DOCK attribute is docked when the window is opened. The following EQUATEs for standard *position* values are contained in EQUATES.CLW:

DOCK:Left	EQUATE(1)
DOCK:Top	EQUATE(2)
DOCK:Right	EQUATE(4)
DOCK:Bottom	EQUATE(8)
DOCK:Float	EQUATE(16)
DOCK:All	EQUATE(31)

Example:

```
Win1 WINDOW('Tools'),TOOLBOX,DOCK(DOCK:All),DOCKED(DOCK:Top) !Dockable anywhere
      BUTTON('Date'),USE(?Button1) !Docked at top on open
      BUTTON('Time'),USE(?Button2)
      END
```

See Also:

DOCK

TOOLBOX

DOCUMENT (create OLE control object from file)**DOCUMENT**(*filename*)

DOCUMENT Specifies creating an object for the OLE control from a data file specific to an OLE server application.

filename A string constant containing the name of the file.

The **DOCUMENT** attribute (PROP:DOCUMENT, write-only) specifies creating an object for the OLE control from a data file specific to an OLE server application. The *filename* parameter syntax must be a fully-qualified pathname, unless the file exists in the same directory as the OLE Controller application.

Example:

```
WinOne WINDOW,AT(0,0,200,200)
      OLE,AT(10,10,160,100),USE(?OLEObject),DOCUMENT('Book1.XLS') !Excel Spreadsheet
      MENUBAR
      MENU('&Clarion App')
      ITEM('&Deactivate Object'),USE(?DeactOLE)
      END
      END
      END
      END
```

DOUBLE, NOFRAME, RESIZE (set window border)**DOUBLE****NOFRAME****RESIZE**

The **DOUBLE**, **NOFRAME**, and **RESIZE** attributes specify a WINDOW or APPLICATION border frame style other than the default single-width border. The **DOUBLE** attribute (PROP:DOUBLE) places a double-width border around the window and the **NOFRAME** attribute (PROP:NOFRAME) places no border on the window. A window with these frame types may not be resized.

The **RESIZE** attribute (PROP:RESIZE) places a thick border frame around the window. This is the only type that allows the user to dynamically resize the window. RESIZE is ignored on any WINDOW with the MODAL attribute.

The RESIZE frame type is normally used on APPLICATION structures and WINDOW structures used as document windows, not dialog boxes. NOFRAME is usually used on "hidden" windows used only to activate an ACCEPT loop. DOUBLE is a common dialog box frame type.

Example:

```
!A Window with a single-width border:
Win1 WINDOW
    END
```

```
!A resizable Window:
Win2 WINDOW,RESIZE
    END
```

```
!A Window with a double-width border:
Win3 WINDOW,DOUBLE
    END
```

```
!A Window without a border:
Win4 WINDOW,NOFRAME
    END
```

DRAGID (set drag-and-drop host signatures)

DRAGID(*signature* [, *signature*])

DRAGID Specifies a LIST or REGION control that can serve as a drag-and-drop host.
signature A string constant containing an identifier used to indicate valid drop targets. Any *signature* that begins with a tilde (~) indicates that the information can also be dragged to an external (Clarion) program. A single DRAGID may contain up to 16 *signatures*.

The **DRAGID** attribute (PROP:DRAGID, an array) specifies a LIST or REGION control that can serve as a drag-and-drop host. DRAGID works in conjunction with the DROPID attribute. The DRAGID *signature* strings (up to 16) define validation keys to match against the *signature* parameters of the target control's DROPID. This provides control over where successful drag-and-drop operations are allowed.

A drag-and-drop operation occurs when the user drags information from a control with the DRAGID attribute to a control with the DROPID attribute. For a successful drag-and-drop operation, both controls must have at least one identical *signature* string in their respective DRAGID and DROPID attributes.

Example:

```
WinOne WINDOW,AT(0,0,160,400)
    !Allows drags, but not drops:
    LIST,AT(120,0,20,20),USE(?List1),FROM(Que1),DRAGID('FromList1')
    !Allows drops from List1, but no drags
    LIST,AT(120,120,20,20),USE(?List2),FROM(Que2),DROPID('FromList1')
END

CODE
OPEN(WinOne)
!Drag and Drop Ids can also be assigned at runtime
?List2{PROP:DropID,1} = 'FromList1'
?List1{PROP:DragID,1} = 'FromList1'
!*****
ACCEPT
CASE EVENT()
OF EVENT:Drag           !When a drag event is attempted
  IF DRAGID()           !check for success
    SETDROPID(Que1)     !and setup info to pass
  END
OF EVENT:Drop           !When drop event is successful
  Que2 = DROPID()       !get dropped info
  ADD(Que2)             !and add it to the queue
END
END
```

See Also: DROPID

DROP (set list box behavior)

DROP(*count* [, *width*])

DROP Specifies the list appears only when the user presses an arrow cursor key or clicks on the drop icon.

count An integer constant that specifies the number of elements displayed.

width An integer constant that specifies the width of the dropped list, in dialog units (PROP:DropWidth, equivalent to {PROP:DROP,2}).

The **DROP** attribute (PROP:DROP) specifies that the selection list appears only when the user presses an arrow cursor key or clicks on the drop icon to the right of the currently selected value display. Once it drops into view, the list displays *count* number of elements. If the DROP attribute is omitted, the LIST or COMBO control always displays the number of data items specified by the *height* parameter of the control's AT attribute in the selection list.

The DROP attribute does not work on a WINDOW with the MODAL attribute and should not be used.

You can assign the name of another icon to the control's PROP:Icon property to override the default down-arrow drop icon.

Example:

```
WinOne WINDOW,AT(0,0,160,400)
    LIST,AT(120,0,20,20),USE(?L7),FROM(Que1),DROP(6)
    COMBO(@S8),AT(120,120,20,20),USE(?C7),FROM(Que2),DROP(8)
END
CODE
OPEN(WinOne)
?C7{PROP:Icon} = 'MyDrop.ICO' !Change the drop icon on the COMBO control
```

DROPID (set drag-and-drop target signatures)

DROPID(*signature* [, *signature*])

DROPID Specifies a control that can serve as a drag-and-drop target.

signature A string constant containing an identifier used to indicate valid drag hosts. A single DROPID may contain up to 16 *signatures*. Any *signature* that begins with a tilde (~) indicates that the information can also be dropped from an external (Clarion) program. A DROPID *signature* of '~FILE' indicates the target accepts a comma-delimited list of filenames dragged from the Windows File Manager.

The **DROPID** attribute (PROP:DROPID, an array) specifies a control that can serve as a drag-and-drop target. DROPID works in conjunction with the DRAGID attribute. The DROPID *signature* strings (up to 16) define validation keys to match against the *signature* parameters of the host control's DRAGID. This provides control over where successful drag-and-drop operations are allowed.

A drag-and-drop operation occurs when the user drags information from a control with the DRAGID attribute to a control with the DROPID attribute. For a successful drag-and-drop operation, both controls must have at least one identical *signature* string in their respective DRAGID and DROPID attributes.

Example:

```
WinOne WINDOW,AT(0,0,160,400)
    !Allows drags, but not drops:
    LIST,AT(120,0,20,20),USE(?List1),FROM(Que1),DRAGID('FromList1')
    !Allows drops from List1 or the Windows File Manager, but no drags:
    LIST,AT(120,120,20,20),USE(?List2),FROM(Que2),DROPID('FromList1','~FILE')
END

CODE
OPEN(WinOne)
!Drag and Drop Ids can also be assigned at runtime
?List2{PROP:DropID,1} = 'FromList1'
?List1{PROP:DragID,1} = 'FromList1'
!*****

ACCEPT
CASE EVENT()
OF EVENT:Drag          !When a drag event is attempted
    IF DRAGID()        !check for success
        SETDROPID(Que1) !and setup info to pass
    END
OF EVENT:Drop          !When drop event is successful
    Que2 = DROPID()    !get dropped info
    ADD(Que2)          !and add it to the queue
END
END
```

See Also: DRAGID

FILL (set fill color)**FILL**(*rgb*)

FILL	Specifies the fill color of a BOX or ELLIPSE control.
<i>rgb</i>	A LONG or ULONG integer constant containing the red, green, and blue components that create the color in the three low-order bytes (bytes 0, 1, and 2) or an EQUATE for a standard Windows color value.

The **FILL** attribute (PROP:FILL) specifies the display or print fill color of a BOX or ELLIPSE control. If omitted, the control is not filled with color. PROP:FILL returns COLOR:None if the FILL attribute is absent.

Example:

```

CustRpt   REPORT,AT(1000,1000,6500,9000),THOUS
CustDetail  DETAIL,AT(0,0,6500,1000)
           ELLIPSE,AT(60,60,200,200),FILL(COLOR:ACTIVEBORDER)  !Color EQUATE
           BOX,AT(360,60,200,200),FILL(00FF0000h)              !Pure Red
           END
           END

WinOne WINDOW,AT(0,0,160,400)
           !Windows' active border color
           BOX,AT(20,20,20,20),FILL(COLOR:ACTIVEBORDER)
           BOX,AT(100,100,20,20),FILL(00FF0000h)  !Blue
           BOX,AT(140,140,20,20),FILL(0000FF00h)  !Green
           BOX,AT(180,180,20,20),FILL(000000FFh)  !Red
           END

```

FIRST, LAST (set MENU or ITEM position)**FIRST****LAST**

The **FIRST** and **LAST** attributes (PROP:FIRST and PROP:LAST) specify menu selection positioning within the global pulldown menu, when a WINDOW's MENUBAR is merged into the global menu. The order of priorities is:

1. Global selections with FIRST attribute
2. Local selections with FIRST attribute
3. Global selections without FIRST or LAST attributes
4. Local selections without FIRST or LAST attributes
5. Global selections with LAST attribute
6. Local selections with LAST attribute

FLAT (set flat control)

FLAT

The **FLAT** attribute (PROP:FLAT) specifies the BUTTON, CHECK, or RADIO with an ICON attribute appears flat until the mouse cursor passes over it. This attribute is typically used on controls placed in a TOOLBAR.

This feature works best if the ICON attribute names a .GIF file to display, as the image will automatically be "grayed" when the control is not active (the mouse cursor is not directly over the control).

The **FLAT** attribute is also supported in LIST and COMBO controls. Activating FLAT for these controls results in a "flatter" appearance (removes the recessed appearance).

PROP:FLAT is a read/write property.

Example:

```
WinOne WINDOW,AT(0,0,160,400)
  TOOLBAR
    CHECK('1'),AT(0,0,20,20),USE(C1),ICON('Check1.GIF'),FLAT
    BUTTON,AT(120,0,20,20),USE(?B7),ICON('Button1.GIF')
    OPTION('Option 4'),USE(OptVar4)
    RADIO('Radio 7'),AT(120,0,20,20),USE(?R7),ICON('Radio1.GIF'),FLAT
    RADIO('Radio 8'),AT(140,0,20,20),USE(?R8),ICON('Radio2.GIF'),FLAT
  END
END
END

CODE
OPEN(WinOne)
?B7{PROP:FLAT} = TRUE
```

FONT (set default font)

FONT([*typeface*] [,*size*] [,*color*] [,*style*] [,*charset*])

FONT	Specifies the default display font for the TOOLBAR .
<i>typeface</i>	A string constant containing the name of the font (PROP:FontName, equivalent to {PROP:Font,1}). If omitted, the system font is used.
<i>size</i>	An integer constant containing the size (in points) of the font (PROP:FontSize, equivalent to {PROP:Font,2}). If omitted, the system default font size is used.
<i>color</i>	A LONG integer constant containing the red, green, and blue values for the color of the font in the low-order three bytes, or an EQUATE for a standard Windows color value (PROP:FontColor, equivalent to {PROP:Font,3}). If omitted, black is used.
<i>style</i>	An integer constant or constant expression or EQUATE specifying the strike weight and style of the font (PROP:FontStyle, equivalent to {PROP:Font,4}). If omitted, the weight is GDI default.
<i>charset</i>	An integer constant or constant expression or EQUATE specifying the character set for the specified FONT (PROP:FontCharSet) or (PROP:Font,5). This provides support for international character sets. If omitted, the default character set for the FONT is used.

The **FONT** attribute (PROP:FONT) specifies the default display font for controls. When the property assignment's *target* is the SYSTEM built-in variable, PROP:FONT sets the font for the MESSAGE procedure.

The *typeface* parameter may name any font registered in the Windows system. For a report, the printer driver must support the specified *typeface* (this includes the TrueType fonts for most printers).

The EQUATES.CLW file contains EQUATE values for standard *style* values. A *style* in the range zero (0) to one thousand (1000) specifies the strike weight of the font. You may add to that values that indicate italic, underline, or strikeout text. The following EQUATES are in EQUATES.CLW:

FONT:thin	EQUATE (100)
FONT:regular	EQUATE (400)
FONT:bold	EQUATE (700)
FONT:italic	EQUATE (01000H)
FONT:underline	EQUATE (02000H)
FONT:strikeout	EQUATE (04000H)

The *charset* parameter may name any *character set* registered in the Windows system. For a report, the printer driver must support the specified *character set*. The EQUATES.CLW file contains EQUATE values for standard *character set* values. The SYSTEM Property, can be used to set the Character Set property at the system level.

CHARSET:ANSI	EQUATE (0)
CHARSET:DEFAULT	EQUATE (1)
CHARSET:SYMBOL	EQUATE (2)
CHARSET:MAC	EQUATE (77)
CHARSET:SHIFTJIS	EQUATE (128)
CHARSET:HANGEUL	EQUATE (129)
CHARSET:JOHAB	EQUATE (130)
CHARSET:GB2312	EQUATE (134)
CHARSET:CHINESEBIG5	EQUATE (136)
CHARSET:GREEK	EQUATE (161)
CHARSET:TURKISH	EQUATE (162)
CHARSET:HEBREW	EQUATE (177)
CHARSET:ARABIC	EQUATE (178)
CHARSET:BALTIC	EQUATE (186)
CHARSET:CYRILLIC	EQUATE (204)
CHARSET:THAI	EQUATE (222)
CHARSET:EASTEUROPE	EQUATE (238)
CHARSET:OEM	EQUATE (255)

Window Usage

The FONT attribute on a WINDOW or APPLICATION structure specifies the default display font for all controls in the WINDOW or APPLICATION that do not have a FONT attribute. This is also the default font for newly created controls on the window, and is the font used by the SHOW and TYPE statements when writing to the window.

The FONT attribute on a TOOLBAR structure specifies the default display font for all controls in the TOOLBAR that do not have a FONT attribute.

Setting any of the runtime properties (PROP:*property*) of the FONT attribute for the WINDOW, APPLICATION, or TOOLBAR does not affect the existing controls already displayed. Controls CREATED after the property has been reset are affected, however.

The FONT attribute on a control declaration overrides any FONT specified on the WINDOW, APPLICATION, or TOOLBAR.

Report Usage

The FONT attribute on a REPORT structure specifies the default print font for all controls in the REPORT. This font is used when the control does not have its own FONT attribute and the print structure containing the control also has no FONT attribute.

The FONT attribute on FORM, DETAIL, HEADER, and FOOTER structures specifies the default print font for all controls in the structures that do not have a FONT attribute.

The FONT attribute on a control declaration overrides any FONT specified on the REPORT or print structure.

Example:

```
LRFont WINDOW('LR FontExample'),AT(,,289,192),|
  FONT('Arial',12,COLOR:Maroon,FONT:bold+FONT:italic,CHARSET:ANSI),GRAY
  !14 point Arial typeface, Red, normal:
  LIST,AT(120,0,20,20),USE(?L7),FROM(Que1),FONT('Arial',14,0FFh)
  !14 point Arial typeface, Black, Bold:
  LIST,AT(120,120,20,20),USE(?C7),FROM(Que2),FONT('Arial',14,0,700)
  !14 point Arial typeface, Black, Bold Italic
  LIST,AT(120,240,20,20),USE(?C7),FROM(Que2),FONT('Arial',14,0,700+01000h)
END
CustRpt REPORT,AT(1000,1000,6500,9000),THOUS, |
  FONT('Arial',12,,FONT:Bold+FONT:Italic)
  !report declarations
END
!A Window using 14 point Times New Roman, Bold and Italic
Win WINDOW,FONT('Times New Roman',14,00H,FONT:italic+FONT:bold)
  STRING('This is Times 14 pt Bold Italic'),AT(42,14),USE(?String1)
END
CODE
OPEN(Win)
Win{PROP:FontSize} = 20           !Set default font size for CREATED controls
CREATE(100,CREATE:string)         !Create a control
100{PROP:Text} = 'This is 20 point'
SETPOSITION(100,82,24)
UNHIDE(100)
ACCEPT
END
```

See Also:

SETFONT, GETFONT, FONTDIALOG, FONTDIALOGA, COLOR, CREATE

FORMAT (set LIST or COMBO layout)**FORMAT**(*format string*)**FORMAT** Specifies the display or print format of the data in the LIST or COMBO control.*format string* A string constant specifying the display format.The **FORMAT** attribute (PROP:FORMAT) specifies the display format of the data in the LIST or COMBO control. The *format string* contains the information for formatting of data.

PROP:FORMAT is updated whenever the user dynamically changes the format of the LIST or COMBO at runtime.

The *format string* contains "field-specifiers" which map to the fields of the QUEUE being displayed. Multiple "field-specifiers" may be grouped together as a "field-group" in square brackets ([]) to display as a single unit.Only the fields in the QUEUE for which there are "field-specifiers" are included in the display. This means that, if there are two fields specified in the *format string* and three fields in the QUEUE, only the two specified in the *format string* are displayed in the LIST or COMBO control.**"Field-specifier" format:**

Each column in the LIST is formatted with the following components. The format for a particular column is returned or set by PROPLIST:Format.

width justification [(*indent*)] [*modifiers*]*width*

A required integer defining the width of the field (PROPLIST:Width). Specified in dialog units.

*justification*A single capital letter (**L**, **R**, **C**, or **D**) that specifies **Left** (PROPLIST:Left), **Right** (PROPLIST:Right), **Center** (PROPLIST:Center), or **Decimal** (PROPLIST:Decimal) justification. One is required.*indent*An optional integer, enclosed in parentheses, that specifies the indent from the justification. This may be negative. With left (**L**) (PROPLIST:LeftOffset) justification, *indent* defines a left margin ; with right (**R**) (PROPLIST:RightOffset) or decimal (**D**) (PROPLIST:DecimalOffset), it defines a right margin; and with center (**C**) (PROPLIST:CenterOffset), it defines an offset from the center of the field (negative = left offset).

modifiers:

Optional special characters (listed below) to modify the display format of the field or group. Multiple *modifiers* may be used on one field or group.

Modifiers:

- * An asterisk (PROPLIST:Color) indicates color information for the field is contained in four LONG fields that immediately follow the data field in the QUEUE (or FROM attribute string). The four colors are normal foreground, normal background, selected foreground, and selected background (in that order). Not valid in a REPORT.
- Y A Y (PROPLIST:CellStyle) indicates a pre-defined style for the field (column) is contained in a LONG field that immediately follows the data field in the QUEUE (or FROM attribute string). The LONG field contains a number that refers to an entry in an array of styles associated with the LIST control through the PROPSTYLE: runtime properties (see below). Not valid in a REPORT.

The style for an entire column may be set with PROPLIST:ColStyle. Using PROPLIST:ColStyle, the LONG field is not necessary in the QUEUE but without the LONG field you cannot assign different styles to individual cells in the column.

- I An I (PROPLIST:Icon) indicates an icon displays in the column, at the left edge of the column (prepended to the data). An icon number is contained in a LONG field immediately following the data field in the QUEUE (or FROM attribute string). The LONG field contains a number that refers to an entry in a list of icons associated with the LIST control through the PROP:IconList runtime property. If an asterisk is also specified for color, this LONG must follow all the color information. Not valid in a REPORT.
- J A J (PROPLIST:IconTrn) indicates a transparent icon displays in the column. The same information as I applies to J. Not valid in a REPORT.

T [(*suppress*)]

A T (PROPLIST:Tree) indicates the LIST is a tree control. The tree level is contained in a LONG field that immediately follows the data field in the QUEUE (or FROM attribute string). If * and I are also specified, this LONG must follow all their LONG fields. The expanded/contracted state of the tree level is determined by the sign of the tree level LONG field's value (positive value=expanded and negative value=contracted). Not valid in a REPORT.

The optional *suppress* parameter can contain a **1** (PROPLIST:TreeOffset) to indicate the root is level number one (1) instead of zero (0), allowing -1 to indicate a contracted root. It can also contain an **R** (PROPLIST:TreeRoot) to suppress the connecting lines to the root level, an **L** (PROPLIST:TreeLines) to suppress the connecting lines between all levels, a **B** (PROPLIST:TreeBoxes) to suppress expansion boxes, and an **I** (PROPLIST:TreeIndent) to suppress level indentation (which also implicitly suppresses both lines and boxes).

~header~ [justification [(indent)]]

A header string enclosed in tildes (PROPLIST:Header), followed by optional justification parameter (**L** = PROPLIST:HeaderLeft, **R** = PROPLIST:HeaderRight, **C** = PROPLIST:HeaderCenter, or **D** = PROPLIST:HeaderDecimal,) and/or indent value in parentheses (PROPLIST:HeaderLeftOffset, PROPLIST:HeaderRightOffset, PROPLIST:HeaderCenterOffset, or PROPLIST:HeaderDecimalOffset), displays the header at the top of the list. The header uses the same justification and indent as the field, if not specifically overridden.

@picture@

The *picture* (PROPLIST:Picture) formats the field for display. The trailing @ is required to define the end of the *picture*, so that display pictures such as @N12~Kr~ can be used in the format string without creating ambiguity.

- ? A question mark (PROPLIST:Locator) defines the locator field for a COMBO list box with a selector field. For a drop-down multi-column list box, this is the value displayed in the current-selection box. Not valid in a REPORT.

#number#

The *number* enclosed in pound signs (#) (PROPLIST:FieldNo) indicates the QUEUE field to display. Following fields in the format string without an explicit #number# are taken in order from the fields following the #number# field. For example, #2# on the first field in the format string indicates starting with the second field in the QUEUE, skipping the first. If the number of fields specified in the format string are >= the number of fields in the QUEUE, the format "wraps around" to the start of the QUEUE.

- _ An underscore (PROPLIST:Underline) underlines the field.
- / A slash (PROPLIST>LastOnLine) causes the next field to appear on a new line (only used on a field within a group).
- | A vertical bar (PROPLIST:RightBorder) places a vertical line to the right of the field.
- M** An M (PROPLIST:Resize) allows the field or group of fields to be dynamically resized at runtime. This allows the user to drag the right vertical bar (if present) or right edge of the data area. Not valid in a REPORT.
- F** An F (PROPLIST:Fixed) creates a fixed column in the list that stays on screen when the user horizontally pages through the fields (by the HSCROLL attribute). Fixed fields or groups must be at the start of the list. This is ignored if placed on a field within a group. Not valid in a REPORT.

S(integer)

An S followed by an *integer* (PROPLIST:Scroll) in parentheses adds a scroll bar to the group. The *integer* defines the total number of dialog units to scroll. This allows large fields to be displayed in a small column width. This is ignored if placed on a field within a group. Not valid in a REPORT.

P

A **P** modifier adds a tool tip (PROPLIST:Tip) to the group. The column's tip text by default is derived from the next queue field that follows the queue field used to hold the actual column's data. If the designated queue field is empty, the Q modifier (see below) designates a string value to use as a default tool tip. Also valid with a VLB (Virtual List Box). Not valid in a REPORT.

Q (string)

A **Q** followed by a string (PROPLIST:DefaultTip) designates the default column tip text to be displayed if the value of the designated P modifier is an empty string. Not valid in a REPORT.

"Field-group" format:

[*multiple field-specifiers*] [(*size*)] [*modifiers*]

multiple field-specifiers

A list of field-specifiers contained in square brackets ([]) that cause them to be treated as a single display unit.

size An optional integer, enclosed in parentheses, that specifies the width of the group (PROPLIST:Width). If omitted, the size is calculated from the enclosed fields.

modifiers

The "field-group" *modifiers* act on the entire group of fields. These are the same *modifiers* listed above for a field (except the *, I, T, and #*number*# *modifiers* which are not appropriate to groups). Add PROPLIST:Group to the appropriate field property to affect the group properties.

PROPLIST:GroupNo can be used to return the group number of a target column.

Example:

```
?List{PROPLIST:GroupNo,LOC:COL}
```

returns the group number of the column variable (LOC:COL). A Column not in a group is considered a group by itself.

```
?List{PROPLIST:GroupNo + PROPLIST:Group,LOC:COL}
```

returns the number of columns in the target group, 0 if the group is really a column.

Display QUEUE Field Format

The order of fields that appear in the QUEUE to display in the LIST is important. Since there are several modifiers which require separate fields in the QUEUE to hold formatting data, the following is the order in which those fields must appear in the QUEUE:

1. The field containing the data to display (always).
2. The * flag's foreground color field (if the * is present, or PROPLIST:Color is set).
3. The * flag's background color field (if the * is present, or PROPLIST:Color is set).
4. The * flag's selected foreground color field (if the * is present, or PROPLIST:Color is set).
5. The * flag's selected background color field (if the * is present, or PROPLIST:Color is set).
6. The **I** or **J** flag's icon field (if the I or J is present, or PROPLIST:Icon or PROPLIST:IconTrn is set).
7. The **T** flag's tree level field (if the T is present, or PROPLIST:Tree is set).
8. The **Y** flag's style field (if the Y is present, or PROPLIST:CellStyle is set).

FORMAT() Runtime Properties

The properties of the individual fields and groups in a multi-column LIST or COMBO control can also be set using the property equates for each (the PROPLIST:Item listed above for each property). These properties eliminate the need to create a complete FORMAT attribute string just to change a single property of a single field in the LIST.

These are all property arrays that require an explicit array element number following the property equate (separated by a comma) to specify which column in the LIST or COMBO is affected. All of them contain blank (") if missing, and a one (1) if present.

Example:

```

PROGRAM
MAP
DisplayList      PROCEDURE
PrintList        PROCEDURE
RandomAlphaData PROCEDURE(*STRING)
END

TreeDemo  QUEUE                !Data list box FROM queue
FName     STRING(20)
ColorNFG  LONG(COLOR:White)  !Normal Foreground color for FName
ColorNBG  LONG(COLOR:Maroon) !Normal Background color for FName
ColorSFG  LONG(COLOR:Yellow) !Selected Foreground color for FName
ColorSBG  LONG(COLOR:Blue)   !Selected Background color for FName
IconField LONG                 !Icon number for FName
TreeLevel LONG                !Tree Level
LName     STRING(20)
Init      STRING(4)
END

CODE
DisplayList
PrintList
DisplayList      PROCEDURE
Win WINDOW('List Boxes'),AT(0,0,366,181),SYSTEM,DOUBLE
LIST,AT(0,34,366,146),FROM(TreeDemo),USE(?Show),HVSCROLL, |
FORMAT('80L*IT~First Name~*80L~Last Name~16C~Initials~')
END

CODE
LOOP X# = 1 TO 20
RandomAlphaData(TreeDemo.FName)
TreeDemo.IconField = ((X#-1) % 4) + 1      !Assign icon number
TreeDemo.TreeLevel = ((X#-1) % 4) + 1      !Assign tree level
RandomAlphaData(TreeDemo.LName)
RandomAlphaData(TreeDemo.Init)
ADD(TreeDemo)
END

```

```

OPEN(Win)
?Show{PROP:iconlist,1} = ICON:VCRback      !Icon 1 = <
?Show{PROP:iconlist,2} = ICON:VCRrewind    !Icon 2 = <<
?Show{PROP:iconlist,3} = ICON:VCRplay     !Icon 3 = >
?Show{PROP:iconlist,4} = ICON:VCRfastforward !Icon 4 = >>
ACCEPT
END

RandomAlphaData PROCEDURE(Field)          !MAP Prototype is: RandomAlphaData(*STRING)
CODE
CLEAR(Field)
RandomSize# = RANDOM(1,SIZE(Field)) !Random fill size
Field[1] = CHR(RANDOM(65,90))         !Start with a random upper case letter
LOOP Z# = 2 to RandomSize#           !Fill each character with
  Field[Z#] = CHR(RANDOM(97,122))     !a random lower case letter
END

PrintList PROCEDURE
DemoQ      QUEUE
FName      STRING(20)
ColorNFG1  LONG
ColorNBG1  LONG
ColorSFG1  LONG(COLOR:Black) !Printed Foreground color for FName
ColorSBG1  LONG(COLOR:White) !Printed Background color for FName
LName      STRING(20)
ColorNFG2  LONG
ColorNBG2  LONG
ColorSFG2  LONG(COLOR:Black) !Printed Foreground color for LName
ColorSBG2  LONG(COLOR:White) !Printed Background color for LName
Init       STRING(4)
ColorNFG3  LONG
ColorNBG3  LONG
ColorSFG3  LONG(COLOR:Black) !Printed Foreground color for Init
ColorSBG3  LONG(COLOR:White) !Printed Background color for Init
Wage       REAL
ColorNFG4  LONG
ColorNBG4  LONG
ColorSFG4  LONG(COLOR:Black) !Printed Foreground color for Wage
ColorSBG4  LONG(COLOR:White) !Printed Background color for Wage
END

CustRpt    REPORT,AT(1000,1000,6500,9000),THOUS
CustDetail DETAIL,AT(0,0,6500,200)
          LIST,AT(0,0,6000,200),FORMAT(' '),FROM(DemoQ),USE(?Show)
          END
          END

```

```
CODE
LOOP X# = 1 TO 20
  CLEAR(DemoQ)
  RandomAlphaData(DemoQ.FName)
  RandomAlphaData(DemoQ.LName)
  RandomAlphaData(DemoQ.Init)
  DemoQ.Wage = RANDOM(100000,1000000)/100
  ADD(DemoQ)
END
OPEN(CustRpt)
SETTARGET(CustRpt)
IF RANDOM(0,1)
  ?Show{PROP:format} = '2000L*~First Name~2000L*~Last Name~500L*~Intls~1000L*~Wage~|'
ELSE
  ?Show{PROP:format} = '2000L*~First Name~2000L*~Last Name~500L*~Intls~1000D(400)*~Wage
  ?Show{PROPLIST:Header,1} = 'First Field'      !Change first field's header text
  ?Show{PROPLIST:Header + PROPLIST:Group,1} = 'First Group'
END
                                !Change first group's header text
LOOP X# = 1 TO RECORDS(DemoQ)
  GET(DemoQ,X#)
  PRINT(CustDetail)
END
CLOSE(CustRpt)
FREE(DemoQ)
```

FORMAT() Style Properties

The following properties are used to set up the array of styles available to a column with the Y modifier or for use with PROPLIST:ColStyle. These define an array of available styles. Assigning the array element number to the Y modifier's LONG field or to PROPLIST:ColStyle sets the display style for the individual cell or column.

PROPSTYLE:FontName

An array property that sets or returns the font name for the style number specified as the array element.

PROPSTYLE:CharSet

An array property that sets or returns the font character set for the style number specified as the array element.

PROPSTYLE:FontSize

An array property that sets or returns the font size for the style number specified as the array element.

PROPSTYLE:FontColor

An array property that sets or returns the font color for the style number specified as the array element.

PROPSTYLE:FontStyle

An array property that sets or returns the font style (strike weight, etc.) for the style number specified as the array element.

PROPSTYLE:TextColor

An array property that sets or returns the text color for the style number specified as the array element (same as fontcolor).

PROPSTYLE:BackColor

An array property that sets or returns the background color for the style number specified as the array element.

PROPSTYLE:TextSelected

An array property that sets or returns the selected text color for the style number specified as the array element.

PROPSTYLE:BackSelected

An array property that sets or returns the selected background color for the style number specified as the array element.

PROPSTYLE:Picture

An array property that sets or returns the display picture associated with the style number specified as the array element.

Example:

```
?list{PROPSTYLE:FontName, 1}      = 'Arial'      !setup positive value style
?list{PROPSTYLE:FontSize, 1}      = 11
?list{PROPSTYLE:FontStyle, 1}     = FONT:Regular
?list{PROPSTYLE:TextColor, 1}     = COLOR:Yellow
?list{PROPSTYLE:BackColor, 1}     = COLOR:Black
?list{PROPSTYLE:TextSelected, 1}  = COLOR:Yellow
?list{PROPSTYLE:BackSelected, 1}  = COLOR:Blue
?list{PROPSTYLE:Picture, 1}       = '@n11.2'

?list{PROPSTYLE:FontName, 2}      = 'Arial'      !setup negative value style
?list{PROPSTYLE:FontSize, 2}      = 11
?list{PROPSTYLE:FontStyle, 2}     = FONT:Bold
?list{PROPSTYLE:TextColor, 2}     = COLOR:Red
?list{PROPSTYLE:BackColor, 2}     = COLOR:White
?list{PROPSTYLE:TextSelected, 2}  = COLOR:Red
?list{PROPSTYLE:BackSelected, 2}  = COLOR:Yellow
?list{PROPSTYLE:Picture, 2}       = '@n(13.2)'

?list{PROPLIST:ColStyle,1} = 1           !Column 1 uses the positive style
?list{PROPLIST:ColStyle,2} = 2           !Column 2 uses the negative style
```

FORMAT() Other List Box Properties

The following properties are not part of the FORMAT attribute string, but may be used to dynamically affect the appearance of the LIST or COMBO control.

PROPLIST:BackColor

An array property that sets or returns the default background color for the text in the column number specified as the array element. This coloring can be overridden on a per-cell basis by the standard cell coloring mechanism.

PROPLIST:BackSelected

An array property that sets or returns the default selected background color for the text in the column number specified as the array element. This coloring can be overridden on a per-cell basis by the standard cell coloring mechanism.

PROPLIST:TextColor

An array property that sets or returns the default text color for the text in the column number specified as the array element. This coloring can be overridden on a per-cell basis by the standard cell coloring mechanism.

PROPLIST:TextSelected

An array property that sets or returns the default selected text color for the text in the column number specified as the array element. This coloring can be overridden on a per-cell basis by the standard cell coloring mechanism.

PROPLIST:Exists

An array property that returns:

- TRUE or FALSE if called with the column number > 0. If the column number specified as the array element exists, PROPLIST:Exists returns TRUE (i.e., ?List{PROPLIST:Exists,1} tests whether column 1 exists in the list). This is useful for generic list box processing.
- total number of list box columns if called with the column number 0
- total number of columns in the specified group if called for the group (i.e., - ?List {PROPLIST:Exists + PROPLIST:Group, GroupNumber})

Example:

```
WinView WINDOW('View'),AT(,,340,200),SYSTEM,CENTER
    LIST,AT(0,0,300,200),USE(?List),FROM(Que),FORMAT('80L~F1~80L~F2~80L~F3~')
    END
CODE
OPEN(WinView)
LOOP X# = 1 TO 255
    IF ?List{PROPLIST:Exists,X#} = 1      !If there is a column with this number
        ?List{PROPLIST:TextColor,X#} = COLOR:Red
        ?List{PROPLIST:BackColor,X#} = COLOR:White
        ?List{PROPLIST:TextSelected,X#} = COLOR:Yellow
        ?List{PROPLIST:TextSelected,X#} = COLOR:Blue
    ELSE
        BREAK
    END
END
```

FORMAT() List Box Mouse Click Properties

The following runtime properties return the mouse position within the LIST or COMBO control when pressed or released. They can also be written to, which has no effect except to temporarily change the value that the property returns when next read (within the same ACCEPT loop iteration). This may make coding easier in some circumstances.

- PROPLIST:MouseDownField
Returns the field number when the mouse is pressed.
- PROPLIST:MouseDownRow
Returns the row number when the mouse is pressed.
- PROPLIST:MouseDownZone
Returns the zone number when the mouse is pressed.
- PROPLIST:MouseMoveField
Returns the field number when the mouse is moved.
- PROPLIST:MouseMoveRow
Returns the row number when the mouse is moved.
- PROPLIST:MouseMoveZone
Returns the zone number when the mouse is moved.
- PROPLIST:MouseUpField
Returns the field number when the mouse is released.
- PROPLIST:MouseUpRow
Returns the row number when the mouse is released.
- PROPLIST:MouseUpZone
Returns the zone number when the mouse is released.

The three "Row" properties all return zero (0) for header text and negative one (-1) if below the last displayed item.

Equates for the following Zones are listed on EQUATES.CLW:

- | | |
|--------------------|---|
| LISTZONE:Field | On a field in the LIST |
| LISTZONE:Right | On the field's right border resize zone |
| LISTZONE:Header | On a field or group header |
| LISTZONE:ExpandBox | On an expand box in a Tree |
| LISTZONE:Tree | On the connecting lines of a Tree |
| LISTZONE:Icon | On an icon (Tree or not) |
| LISTZONE:Nowhere | Anywhere else |

Example:

```

Que  QUEUE
F1   STRING(50)
F2   STRING(50)
F3   STRING(50)
      END

WinView WINDOW('View'),AT(,,340,200),SYSTEM,CENTER
      LIST,AT(20,0,300,200),USE(?List),FROM(Que),HVSCROLL, |
      FORMAT('80L~F1~80L~F2~80L~F3~'),ALRT(MouseLeft)
      END

SaveFormat  STRING(20)
SaveColumn  BYTE
Columns     BYTE,DIM(3)

CODE
OPEN(WinView)
Columns[1] = 1
Columns[2] = 2
Columns[3] = 3
DO BuildListQue
ACCEPT
CASE EVENT()
OF EVENT:PreAlertKey
CYCLE                                     !Allow standard LIST clicks to pro
OF EVENT:AlertKey
IF ?List{PROPLIST:MouseDownRow} = 0       !Check for click in header
EXECUTE Columns[?List{PROPLIST:MouseDownField}] !Check which header
SORT(Que,Que.F1)
SORT(Que,Que.F2)
SORT(Que,Que.F3)
END
SaveFormat = ?List{PROPLIST:Format,?List{PROPLIST:MouseDownField}}
?List{PROPLIST:Format,?List{PROPLIST:MouseDownField}} = ?List{PROPLIST:Format,1}
?List{PROPLIST:Format,1} = SaveFormat
SaveColumn = Columns[?List{PROPLIST:MouseDownField}]
Columns[?List{PROPLIST:MouseDownField}] = Columns[1]
Columns[1] = SaveColumn
DISPLAY
END
END
END
FREE(Que)

```

```
BuildListQue ROUTINE
LOOP Y# = 1 TO 9
  Que.F1 = 'Que.F1 - ' & Y#
  Que.F2 = 'Que.F2 - ' & RANDOM(10,99)
  Que.F3 = 'Que.F3 - ' & RANDOM(100,999)
  ADD(Que)
  ASSERT(NOT ERRORCODE())
END
```

FROM (set listbox data source)

FROM(*source*)

FROM	Specifies the source of the data displayed or printed in a LIST control.
<i>source</i>	The label of a QUEUE or field within a QUEUE, or a string constant or variable (normally a GROUP) containing the data items to display or print in the LIST. If the QUEUE has been dynamically created with NEW, the corresponding DISPOSE <u>must</u> come after the window has been closed.

The **FROM** attribute (PROP:FROM, write-only) specifies the source of the data elements displayed in a LIST, COMBO, or SPIN control, or printed in a LIST control.

If a string constant is specified as the *source*, the individual data elements must be delimited by a vertical bar (|) character. To include a vertical bar as part of one data element, place two adjacent vertical bars in the string (||), and only one will be displayed. To indicate that an element is empty, place at least one blank space between the two vertical bars delimiting the elements (| |).

Normally, a valid event will move the selected data element's contents into the control's USE variable. However, an *alternate* value can also be specified in the FROM attribute by including an additional vertical bar followed by a pound symbol (#), and followed by the alternate value.

Window Usage

For a SPIN control, the *source* would usually be a QUEUE field or string. If the *source* is a QUEUE with multiple fields, only the first field is displayed in the SPIN.

For LIST and COMBO controls, the data elements are formatted for display according to the information in the FORMAT attribute. If the label of a QUEUE is specified as the *source*, all fields in the QUEUE are displayed as defined by the FORMAT attribute. If the label of one field in a QUEUE is specified as the *source*, only that field is displayed.

Report Usage

If the label of a QUEUE is specified as the *source*, all fields in the QUEUE are printed. If the label of one field in a QUEUE is specified as the *source*, only that field is printed. Only the current QUEUE entry in the queue's data buffer is printed in the LIST. If a string constant or variable is specified as the *source*, the entire string (all entries in the vertical bar delimited list of data elements) is printed in the LIST. The data elements are formatted for printing in the LIST according to the information in the FORMAT attribute.

Example:

```
TD    QUEUE,AUTO
FName STRING(20)
LName STRING(20)
Init  STRING(4)
Wage  REAL
      END
```

```
CustRpt    REPORT,AT(1000,1000,6500,9000),THOUS
CustDetail DETAIL,AT(0,0,6500,1000)
           LIST,AT(0,34,366,146),FORMAT('80L80L16L60L'),FROM(TD),USE(?Show1)
           LIST,AT(0,200,100,146),FORMAT('80L'),FROM(Fname),USE(?Show2)
           END
           END
```

```
Que1  QUEUE,PRE(Q1)
F1    LONG
F2    STRING(8)
      END
```

```
Win1  WINDOW,AT(0,0,160,400)
      LIST,AT(120,0,20,20),USE(?L1),FROM(Que1),FORMAT('5C~List~15L~Box~'),COLUMN
      COMBO(@S8),AT(120,120,20,20),USE(?C1),FROM(Q1:F2)
      SPIN(@N8.2),AT(280,0,20,20),USE(SpinVar1),FROM(Q1:F1)
      SPIN(@S4),AT(280,0,20,20),USE(SpinVar2),FROM('Mr. |Mrs. |Ms. |Dr. ')
      !Assign a numeric value to SpinVar3:
      SPIN(@S4),AT(280,0,20,20),USE(SpinVar3),FROM('Mr. |#1|Mrs. |#2|Ms. |#3|Dr. |#4')
      END
```

FULL (set full-screen)**FULL**

The **FULL** attribute (PROP:FULL) specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter.

FULL may not be specified for TOOLBAR controls.

GRAY (set 3-D look background)

GRAY

The **GRAY** attribute (PROP:GRAY) indicates that the WINDOW has a gray background, suitable for use with three-dimensional dialog controls. All controls on a WINDOW with the GRAY attribute are automatically given a three-dimensional appearance. Controls in a TOOLBAR are always automatically given a three-dimensional appearance, without the GRAY attribute.

This attribute is not valid on an APPLICATION structure.

The three-dimensional look may be disabled by SET3DLOOK.

Example:

```
!A Window with 3-D controls  
Win1 WINDOW,GRAY  
END
```

See Also:

SET3DLOOK

GRID (set list grid-line display color)**GRID**(*rgb*)

GRID Specifies list box grid-line display color.*rgb* A LONG or ULONG integer constant, or constant EQUATE, containing the red, green, and blue components that create the color in the three low-order bytes (bytes 0, 1, and 2), or an EQUATE for a standard Windows color value.

The **GRID** attribute (PROPLIST:GRID) specifies the display color of grid-lines in a COMBO, or LIST control. EQUATEs for Windows' standard colors are contained in the EQUATES.CLW file. Windows automatically finds the closest match to the specified *rgb* color value for the hardware on which the program is run.

Example:

```
WinOne WINDOW,AT(0,0,400,400)
      LIST,AT(0,34,366,146),FROM(TreeDemo),USE(?Show),HVSCROLL,GRID(COLOR:Red)|
      FORMAT('80L*IT~First Name~*80L~Last Name~16C~Initials~')
END
```

HIDE (set control hidden)**HIDE**

The **HIDE** attribute (PROP:HIDE) specifies the control does not appear when the WINDOW or APPLICATION is first opened. UNHIDE must be used to display it. In a REPORT, the control does not print unless the UNHIDE statement is used to allow it to print. PROP:HIDE may be used with the TARGET built-in variable to hide/unhide the current target window.

HLP (set on-line help identifier)**HLP**(*helpID*)

HLP	Specifies the <i>helpID</i> for the APPLICATION, WINDOW, or control.
<i>helpID</i>	A string constant specifying the key used to access the Help system. This may be either a Help keyword or a "context string."

The **HLP** attribute (PROP:HLP) specifies the *helpID* for the APPLICATION or WINDOW. Help, if available, is automatically displayed by Windows whenever the user presses F1.

If the user presses F1 to request help when the APPLICATION window is foremost and no menus are active, the APPLICATION's *helpID* is used to locate the Help text. Otherwise, the library automatically uses the *helpID* of the active menu of uppermost control or window, searching up the hierarchy until an object with that *helpID* is found. The *helpID* of the APPLICATION is at the top of the hierarchy.

The *helpID* may contain a Help keyword or a "context string."

- A Help keyword is a word or phrase displayed in the Help Search dialog. When the user presses F1, if only one topic in the help file specifies this keyword, the help file is opened at that topic; if more than one topic specifies the keyword, the search dialog is opened for the user.
- A "context string" is identified by a leading tilde (~) in the *helpID*, followed by a unique identifier (no spaces allowed) associated with exactly one help topic. When the user presses F1, the help file is opened at the specific topic associated with that "context string." If the tilde is missing, the *helpID* is assumed to be a help keyword.

Example:

```

!A Window with a help context string:
MainWin APPLICATION('My Application'),SYSTEM,MAX,ICON('MyIcon.ICO'),STATUS,HLP('~App')
    MENUBAR
        MENU('&File'),USE(?FileMenu)
            ITEM('&Open...'),USE(?OpenFile),HLP('~OpenFileHelp')
        END
    END
END

!A Window with a help keyword:
Win2 WINDOW,HLP('Window One Help')
    ENTRY(@s30),USE(SomeVariable),HLP('~Entry1Help')      !A help context string
    ENTRY(@s30),USE(SomeVariable),HLP('Control Two Help') !A help keyword
END

```

HSCROLL, VSCROLL, HVSCROLL (set scroll bars)

HSCROLL

VSCROLL

HVSCROLL

The **HSCROLL**, **VSCROLL**, and **HVSCROLL** attributes place scroll bars on an APPLICATION or WINDOW structure, or a COMBO, LIST, IMAGE, or TEXT control. HSCROLL (PROP:HSCROLL) adds a horizontal scroll bar to the bottom, VSCROLL (PROP:VSCROLL) adds a vertical scroll bar on the right side, and HVSCROLL (PROP:HVSCROLL) adds both.

The HSCROLL attribute is also available for a SHEET control. This specifies the TABs display all on one row instead of multiple rows, no matter how many TABs there are. Right and left (or up and down) scroll buttons appear at both ends of the TABs to allow the user to scroll through all the TABs. PROP:BrokenTabs may be set to FALSE to turn off the "broken tab" visual effect.

The vertical scroll bar allows a mouse to scroll the display up or down. The horizontal scroll bar allows a mouse to scroll the control's display left or right. The scroll bars appear whenever any scrollable portion of the control lies outside the visible area on screen.

When you place VSCROLL on a LIST with the IMM attribute, the vertical scroll bar is always present, even when the list is not full. When the user clicks on the scroll bar, events are generated, but the list contents do not move (executable code should perform this task). You can interrogate the PROP:VscrollPos property to determine the scroll thumb's position in the range 0 (top) to 255 (bottom).

HSCROLL, VSCROLL, and HVSCROLL are also valid on a SPIN control and specify optional spin button arrangements from the default (one above the other, pointing up and down). HSCROLL places the spin buttons side by side pointing left and right, VSCROLL places the spin buttons one above the other pointing left and right, and HVSCROLL places the spin buttons side by side, pointing up and down.

Example:

```
!A Window with a horizontal scroll bar:  
Win1 WINDOW,HSCROLL,RESIZE  
END
```

```
!A Window with a vertical scroll bar:  
Win2 WINDOW,VSCROLL,RESIZE  
END
```

```
!A Window with both scroll bars:  
Win2 WINDOW,HVSCROLL,RESIZE  
END
```

ICON (set icon)**ICON**([*file*])

ICON	Specifies an icon to display for the APPLICATION, WINDOW, or control.
<i>file</i>	A string constant containing the name of the image file (.ICO, .GIF, .JPG, .PCX) or an EQUATE for the Windows standard icon to display. The image file is automatically linked into the .EXE as a resource.

The **ICON** attribute (PROP:ICON) specifies an icon to display for the APPLICATION, WINDOW, or control.

On an APPLICATION or WINDOW, **ICON** also specifies the presence of a minimize control, and must name an .ICO file as its *file* parameter. The minimize control appears in the top right corner of the window as a downward pointing triangle (in Windows 3.1) or an underscore (Windows 95). When the user clicks the mouse on it, the window shrinks to an icon without halting its execution. When an APPLICATION or non-MDI WINDOW is minimized, the icon *file* is displayed in the operating system's desktop; when a WINDOW with the MDI attribute is minimized, the icon *file* is displayed in the APPLICATION's client area.

On a BUTTON, RADIO, or CHECK control, **ICON** specifies an image to display as the control. The image *file* displays on the button face of the control. For RADIO and CHECK controls, the **ICON** attribute creates "latched" pushbuttons, where the control button appears "down" when on and "up" when off.

EQUATE statements for the Windows-standard icons are contained in the EQUATES.CLW file. The following list is a representative sample of these (see EQUATES.CLW for the complete list):

ICON:None	No icon
ICON:Application	
ICON:Question	?
ICON:Exclamation	!
ICON:Asterisk	*
ICON:VCRtop	>>
ICON:VCRrewind	<<
ICON:VCRback	<
ICON:VCRplay	>
ICON:VCRfastforward	>>
ICON:VCRbottom	<<
ICON:VCRlocate	?

If the name of the icon file to assign to PROP:Icon has a number in square brackets appended to its end (IconFile.DLL[1]), this indicates the file contains multiple icons and the number specifies which to assign (using zero-based numbering). If the name of the icon file has a tilde (~) prepended to it (~IconFile.ICO), this indicates the file has been linked into the project as a resource and is not on disk.

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL,ICON('MyIcon.ICO')
    OPTION('Option'),USE(OptVar)
        RADIO('Radio 1'),AT(120,0,20,20),USE(?R1),ICON('Radio1.ICO')
        RADIO('Radio 2'),AT(140,0,20,20),USE(?R2),ICON('Radio2.GIF')
    END
    CHECK('&A'),AT(0,120,20,20),USE(?C7),ICON(ICON:Asterisk)
    BUTTON('&1'),AT(120,0,20,20),USE(?B7),ICON(ICON:Question)
END
```

See Also:

ICONIZE

MAX

MAXIMIZE

IMM

ICONIZE (set window open as icon)

ICONIZE

The **ICONIZE** attribute (PROP:ICONIZE) specifies the APPLICATION or WINDOW is opened minimized as the icon specified by the ICON attribute. When an APPLICATION or non-MDI WINDOW is minimized, the icon *file* is displayed in the operating system's desktop; when a WINDOW with the MDI attribute is minimized, the icon *file* is displayed in the APPLICATION.

With the SYSTEM built-in variable as the property assignment *target*, PROP:ICONIZE returns 1 if the Windows PIF setting for the application is set to open the application iconized.

Example:

```
!A Window with a minimize button, opened as the icon:  
Win2 WINDOW,ICON('MyIcon.ICO'),ICONIZE  
END
```

See Also:

ICON

IMM

IMM (set immediate event notification)

IMM

The **IMM** attribute (PROP:IMM) specifies immediate event generation.

Window Usage

On a WINDOW or APPLICATION the IMM attribute specifies immediate event generation whenever the user moves or resizes the window. It generates one the following events before the action is executed:

- EVENT:Move
- EVENT:Size
- EVENT:Restore
- EVENT:Maximize
- EVENT:Iconize

If the code that handles these events executes a CYCLE statement, the action is not performed. This allows you to prevent the user from moving or resizing the window. Once the action has been performed, one or more of the following events are generated:

- EVENT:Moved
- EVENT:Sized
- EVENT:Restored
- EVENT:Maximized
- EVENT:Iconized

Multiple post-action events generate because some actions have multiple results. For example, if the user CLICKS on the maximize button, EVENT:Maximize generates. If there is no CYCLE statement executed for this event, the action is performed, then EVENT:Maximized, EVENT:Moved, and EVENT:Sized all generate. This occurs because the window was maximized, which also moves and resizes it at the same time.

Control Usage

On a REGION control, the IMM attribute generates an event whenever the mouse enters (EVENT:MouseIn), moves within (EVENT:MouseMove), or leaves (EVENT:MouseOut) the area specified by the REGION's AT attribute. The exact position of the mouse can be determined by the MOUSEX and MOUSEY procedures.

On a BUTTON control, the IMM attribute indicates the BUTTON generates EVENT:Accepted when the left mouse button is pressed down on the control, instead of on its release. EVENT:Accepted repeatedly generates as long as the user keeps the mouse button pressed. The DELAY and REPEAT attributes on the BUTTON can change the rate the events generate.

The IMM attribute specifies immediate event generation each time the user presses any keystroke on a LIST or COMBO control, usually requiring the QUEUE to be re-filled. This means all keys are implicitly ALRTed for the control. When the user presses a character, EVENT:NewSelection generates.

For an ENTRY or SPIN control, EVENT:NewSelection generates whenever the control's contents or the cursor position changes. To do something only when the content changes, you must save the previous contents then compare against the current contents (probably using PROP:ScreenText).

For a SHEET control, EVENT:NewSelection generates whenever the user clicks on a TAB (even when that TAB is already the currently selected TAB). This can be useful when there are multiple SHEET controls on the same window.

Example:

```
Win2 WINDOW('Some Window'),AT(58,11,174,166),MDI,DOUBLE,MAX,IMM
    LIST,AT(109,48,50,50),USE(?List),FROM('Que'),IMM
    BUTTON('&Ok'),AT(111,108,,),USE(?Ok)
    BUTTON('&Cancel'),AT(111,130,,),USE(?Cancel)
END
CODE
OPEN(Win2)
ACCEPT
CASE EVENT()
OF EVENT:Move           !Prevent user from moving window
CYCLE
OF EVENT:Maximized     !When Maximized
?List{PROP:Height} = 100 ! resize the list
OF EVENT:Restored      !When Restored
?List{PROP:Height} = 50 ! resize the list
END
END
```

See Also:

RESIZE

MAX

ICON

DELAY

REPEAT

INS, OVR (set typing mode)**INS****OVR**

The **INS** and **OVR** attributes (PROP:INS and PROP:OVR) specify the typing mode for an ENTRY, SPIN or COMBO control when the MASK attribute is present on the window. INS specifies insert mode while OVR specifies overwrite mode. These modes are only active on windows with the MASK attribute.

JOIN (set joined TAB scroll buttons)**JOIN**

The **JOIN** attribute (PROP:JOIN) on a SHEET control specifies the TABs display all on one row instead of multiple rows, no matter how many TABs there are. Right and left (or up and down) scroll buttons appear together at the right end (or top) of the TABs to allow the user to scroll through all the TABs.

KEY (set execution keycode)*KEY(keycode)*

KEY Specifies a "hot" key for the control

keycode A Clarion Keycode or keycode equate label.

The **KEY** attribute (PROP:KEY) specifies a "hot" key to immediately give focus to the control or execute the control's associated action.

The following controls receive focus: COMBO, CUSTOM, ENTRY, GROUP, LIST, OPTION, PROMPT, SPIN, TEXT.

The following controls both receive focus and immediately execute: BUTTON, CHECK, CUSTOM, RADIO, MENU, ITEM.

Example:

```

MainWin APPLICATION('My Application'),SYSTEM,MAX,ICON('MyIcon.ICO'),STATUS
  MENUBAR
  MENU('&Edit'),USE(?EditMenu)
    ITEM('Cu&t'),USE(?CutText),KEY(CtrlX),STD(STD:Cut),DISABLE
    ITEM('&Copy'),USE(?CopyText),KEY(CtrlC),STD(STD:Copy),DISABLE
    ITEM('&Paste'),USE(?PasteText),KEY(CtrlV),STD(STD:Paste),DISABLE
  END
END
TOOLBAR
COMBO(@S8),AT(120,120,20,20),USE(?C1),FROM(Q1:F2),KEY(F1Key)
LIST,AT(120,0,20,20),USE(?L1),FROM(Que1),KEY(F2Key)
SPIN(@N8.2),AT(280,0,20,20),USE(SpinVar1),FROM(Q),KEY(F3Key)
TEXT,AT(20,0,40,40),USE(E2),KEY(F4Key)
PROMPT('Enter &Data in E2:'),AT(10,200,20,20),USE(?P2),KEY(F5Key)
ENTRY(@S8),AT(100,200,20,20),USE(E2),KEY(F6Key)
BUTTON('&1'),AT(120,0,20,20),USE(?B7),KEY(F7Key)
CHECK('&A'),AT(0,120,20,20),USE(?C7),KEY(F8Key)
OPTION('Option'),USE(OptVar),KEY(F9Key)
RADIO('Radio 1'),AT(120,0,20,20),USE(?R1),KEY(F10Key)
RADIO('Radio 2'),AT(140,0,20,20),USE(?R2),KEY(F11Key)
END
END
END

```

LANDSCAPE (set page orientation)**LANDSCAPE**

The **LANDSCAPE** attribute (PROP:LANDSCAPE) on a REPORT indicates the report is to print in landscape mode by default. If the LANDSCAPE attribute is omitted, printing defaults to portrait mode.

Example:

```
Report REPORT,PRE('Rpt'),LANDSCAPE    !Defaults to landscape mode
      !Report structure declarations
      END
```

LAYOUT (set window orientation)

LAYOUT (*style*)

style A byte value that designates whether the orientation of window controls and field sequence is set to Default, Left to Right, or Right to Left.

The **LAYOUT** attribute (PROP:LAYOUT) on a WINDOW indicates the orientation of window controls and field sequence. A style of (1) essentially "flips" the window controls' display as a mirror image of the layout specified in the Window Formatter. Default field navigation moves from right to left.

A style of (0) maintains the original layout specified in the Window Formatter. Default field navigation moves from left to right.

Setting the LAYOUT in the Application Frame will cascade its setting to all child window that have the default setting active (which is no LAYOUT attribute placed on the window).

Example:

```
AppFrame APPLICATION('Example Application - BROWSES (ABC Templates)'), |
    AT(, ,548,344),FONT('MS Sans Serif',8,,),LAYOUT(1), |
    CENTER,ICON('LOG1.ICO'),STATUS(-1,80,120,45),SYSTEM,MAX,RESIZE
!Sets all windows in the application to Right-to-Left style
```

LEFT, RIGHT, ABOVE, BELOW (set TAB position)**LEFT**([*width*])**RIGHT**([*width*])**ABOVE**([*width*])**BELOW**([*width*])

width An integer constant specifying the width of the TAB controls in dialog units. For the LEFT attribute, this is PROP:LeftOffset (equivalent to {PROP:LEFT,2}). For RIGHT, this is PROP:RightOffset (equivalent to {PROP:RIGHT,2}). For ABOVE, this is PROP:AboveSize (equivalent to {PROP:ABOVE,2}). For BELOW, this is PROP:BelowSize (equivalent to {PROP:BELOW,2}).

The **LEFT**, **RIGHT**, **ABOVE**, and **BELOW** attributes of a SHEET control specify the position of the TAB controls. LEFT (PROP:LEFT) specifies the TABs appear to the left of the sheet, RIGHT (PROP:RIGHT) specifies the TABs appear to the right of the sheet, ABOVE (PROP:ABOVE) specifies the TABs appear at the top of the sheet (the default position), and BELOW (PROP:BELOW) specifies the TABs appear at the bottom of the sheet.

The *width* parameter allows you to set the size of the TAB controls. The text that appears on the TAB is always horizontal unless you specify the UP or DOWN attribute.

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
  SHEET,AT(0,0,320,175),USE(SelectedTab),BELOW !Place Tabs below sheet
  TAB('Tab One'),USE(?TabOne)
    OPTION('Option 1'),USE(OptVar1),KEY(F10Key),HLP('Option1Help')
    RADIO('Radio 1'),AT(20,0,20,20),USE(?R1)
    RADIO('Radio 2'),AT(40,0,20,20),USE(?R2)
  END
  PROMPT('Enter Data:'),AT(100,100,20,20),USE(?P1)
  ENTRY(@S8),AT(100,140,32,20),USE(E1)
  PROMPT('Enter More Data:'),AT(100,200,20,20),USE(?P2)
  ENTRY(@S8),AT(100,240,32,20),USE(E2)
  END
  TAB('Tab Two'),USE(?TabTwo)
    PROMPT('Enter Data:'),AT(100,100,20,20),USE(?P3)
    ENTRY(@S8),AT(100,140,32,20),USE(E3)
    PROMPT('Enter More Data:'),AT(100,200,20,20),USE(?P4)
    ENTRY(@S8),AT(100,240,32,20),USE(E4)
  END
  END
  BUTTON('Ok'),AT(100,180,20,20),USE(?Ok)
  BUTTON('Cancel'),AT(200,180,20,20),USE(?Cancel)
  END
```

LEFT, RIGHT, CENTER, DECIMAL (set justification)**LEFT**([*offset*])**RIGHT**([*offset*])**CENTER**([*offset*])**DECIMAL**([*offset*])

offset An integer constant specifying the amount of offset from the justification point. This is in dialog units (unless overridden by the THOUS, MM, or POINTS attribute on a REPORT). For the LEFT attribute, this is PROP:LeftOffset (equivalent to {PROP:LEFT,2}). For RIGHT, this is PROP:RightOffset (equivalent to {PROP:RIGHT,2}). For CENTER, this is PROP:CenterOffset (equivalent to {PROP:CENTER,2}). For DECIMAL, this is PROP:DecimalOffset (equivalent to {PROP:DECIMAL,2}).

The **LEFT**, **RIGHT**, **CENTER**, and **DECIMAL** attributes specify the justification of data printed. LEFT (PROP:LEFT) specifies left justification, RIGHT (PROP:RIGHT) specifies right justification, CENTER (PROP:CENTER) specifies centered text, and DECIMAL (PROP:DECIMAL) specifies numeric data aligned on the decimal point.

On the LEFT attribute, *offset* specifies the amount of indentation from the left. On the RIGHT attribute, *offset* specifies the amount of indentation from the right. The *offset* parameter on the CENTER attribute specifies an offset value from the center (negative = left offset). On the DECIMAL attribute, *offset* specifies the decimal point's indentation from the right.

Window Usage

The following controls allow LEFT or RIGHT only (without an *offset* parameter): BUTTON, CHECK, RADIO.

The following controls allow LEFT(*offset*), RIGHT(*offset*), CENTER(*offset*), or DECIMAL(*offset*): COMBO, ENTRY, LIST, SPIN, STRING.

The TEXT control allows LEFT(*offset*), RIGHT(*offset*), or CENTER(*offset*).

Report Usage

The following controls allow LEFT or RIGHT only (without an *offset* parameter): CHECK, GROUP, OPTION, RADIO.

The following controls allow LEFT(*offset*), RIGHT(*offset*), CENTER(*offset*), or DECIMAL(*offset*): LIST, STRING.

The TEXT control allows LEFT, RIGHT, and CENTER (without an *offset* parameter).

Example:

```
Rpt  REPORT,AT(1000,1000,6500,9000),THOUS
Detail DETAIL,AT(0,0,6500,1000)
      LIST,AT(0,20,100,146),FORMAT('800L'),FROM(Fname),USE(?Show2),LEFT(100)
      END
      END

WinOne WINDOW,AT(0,0,160,400)
      COMBO(@S8),AT(120,120,20,20),USE(?C1),FROM(Q1:F2),RIGHT(4)
      LIST,AT(120,0,20,20),USE(?L1),FROM(Que1),CENTER
      SPIN(@N8.2),AT(280,0,20,20),USE(SpinVar1),FROM(Q),DECIMAL(8)
      TEXT,AT(20,0,40,40),USE(E2),LEFT(8)
      ENTRY(@S8),AT(100,200,20,20),USE(E2),LEFT(4)
      CHECK('&A'),AT(0,120,20,20),USE(?C7),LEFT
      OPTION('Option'),USE(OptVar)
      RADIO('Radio 1'),AT(120,0,20,20),USE(?R1),LEFT
      RADIO('Radio 2'),AT(140,0,20,20),USE(?R2),RIGHT
      END
      END
```

LINEWIDTH (set line thickness)**LINEWIDTH**(*width*)**LINEWIDTH** Specifies the LINE control and BOX and ELLIPSE border thickness.*width* A positive integer constant specifying the thickness in pixels.The **LINEWIDTH** attribute (PROP:LINEWIDTH) specifies the thickness of the LINE control and the BOX and ELLIPSE controls' border.

Example:

```

CustRpt REPORT,AT(1000,1000,6500,9000),THOUS
Detail  DETAIL,AT(0,0,6500,1000),USE(?DetailOne)
        LINE,AT(105,78,-49,0),USE(?Line1),LINEWIDTH(3) !3 pixel line
        BOX,AT(182,27,50,50),USE(?Box1),LINEWIDTH(3)   !Box with 3 pixel border
        STRING(@N$11.2),AT(6000,1500,500,500),USE(Pre:F1)
        END
      END

window WINDOW('Caption'),AT(,,260,100),GRAY
        LINE,AT(105,78,-49,0),USE(?Line1),LINEWIDTH(3) !3 pixel line
        BOX,AT(182,27,50,50),USE(?Box1),LINEWIDTH(3)   !Box with 3 pixel border
      END

```

LINK (create OLE control link to object from file)

LINK(*filename*)

LINK Specifies creating a link to an object for the OLE control from a data file specific to an OLE server application.

filename A string constant containing the name of the file.

The **LINK** attribute (PROP:LINK, write-only) specifies creating a link to an object for the OLE control from a data file specific to an OLE server application. The *filename* parameter syntax must be a fully-qualified pathname, unless the file exists in the same directory as the OLE Controller application.

Example:

```
WinOne WINDOW,AT(0,0,200,200)
  OLE,AT(10,10,160,100),USE(?OLEObject),LINK('Book1.XLS') !Excel Spreadsheet
  MENUBAR
  MENU('&Clarion App')
  ITEM('&Deactivate Object'),USE(?DeactOLE)
END
END
END
END
```

MARK (set multiple selection mode)

MARK(*flag*)

MARK Enables multiple items selection.

flag The label of a QUEUE field.

The **MARK** attribute (PROP:MARK, write-only) enables multiple item selections from a LIST or COMBO control. When an item in the LIST is selected, the appropriate *flag* field is set to true (1). Each marked entry is automatically highlighted in the LIST or COMBO. Changing the value of the *flag* field also changes the screen display for the related LIST or COMBO entry.

If the MARK attribute is specified on the LIST or COMBO, the IMM attribute may not be.

Example:

```

Que1    QUEUE
MarkFlag BYTE
F1      LONG
F2      STRING(8)
        END

```

```

WinOne WINDOW,AT(0,0,160,400),SYSTEM
        LIST,AT(120,0,20,20),USE(?L1),FROM(Que1.F1),MARK(Que1.MarkFlag)
        COMBO(@S8),AT(120,120,,),USE(?C1),FROM(Que1.F2),MARK(Que1.MarkFlag)
        END

```

```

CODE
DO LoadQue           !Load Que1 with data
OPEN(WinOne)
ACCEPT
CASE EVENT()
OF EVENT:CloseWindow
  BREAK
END
END
LOOP X# = 1 to RECORDS(Que1) !Loop through queue entries
  GET(Que1,X#)
  IF Que1.MarkFlag           !If the user marked the entry
    DO ProcessMarked         ! then process it
  END
END
END

```

MASK (set pattern editing data entry)**MASK**

The **MASK** attribute (PROP:MASK) specifies pattern input editing mode for all controls in the window (when placed on the WINDOW statement). Toggling the value of PROP:MASK for a window only affects controls created after--it does not affect any existing controls.

Pattern input editing mode means that, as the user types in data, each character is automatically validated against the control's picture for proper input (numbers only in numeric pictures, etc.). This forces the user to enter data in the format specified by the control's display picture.

If MASK is omitted, Windows free-input is allowed in the controls. Free-input means the user's data is formatted to the control's picture only after entry. This allows users to enter data as they choose and it is automatically formatted to the control's picture after entry. If the user types in data in a format different from the control's picture, the libraries attempt to determine the format the user used, and convert the data to the control's display picture. For example, if the user types "January 1, 1995" into a control with a display picture of @D1, the runtime library formats the user's input to "1/1/95." This action occurs only after the user completes data entry and moves to another control. If the runtime library cannot determine what format the user used, it will not update the USE variable. It then beeps and leaves the user on the same control with the data they entered, to allow them to try again.

Example:

```
!A Window with pattern input editing enabled
Win2 WINDOW, MASK
    END

!Window with controls with pattern input editing enabled
Win2 WINDOW, MASK
    COMBO(@P(###) ###-####P), AT(120, 120, 20, 20), USE(Phone), FROM(Q1:F2)
    SPIN(@N8.2), AT(280, 0, 20, 20), USE(SpinVar1), FROM(Q)
    ENTRY(@D2), AT(100, 200, 20, 20), USE(DateField)
    END
```

MAX (set maximize control or total maximum)

MAX([*variable*])

MAX	Specifies a maximize control on an APPLICATION or WINDOW, or calculates the maximum value a REPORT STRING control's USE variable has contained so far.
<i>variable</i>	The label of a numeric variable to receive the intermediate values calculated for the MAX (valid only in a REPORT). This allows you to create totals on other totals. The value in the <i>variable</i> is internally updated by the print engine, so it is only useful for use by other "totaling" controls within the REPORT structure.

The **MAX** attribute (PROP:MAX) specifies a maximize control on an APPLICATION or WINDOW, or calculates the maximum value a REPORT STRING control's USE variable has contained so far.

Window Usage

The maximize control appears in the top right corner of the window as a box containing either an upward pointing triangle, or an upward pointing triangle above a downward pointing triangle (in Windows 3.1). When the user clicks the mouse on it, an APPLICATION or non-MDI WINDOW expands to occupy the full screen, an MDI WINDOW expands to occupy the entire APPLICATION. Once expanded, the maximize control appears as an upward pointing triangle above a downward pointing triangle. Click the mouse on it again, and the window returns to its previous size and the maximize control appears as an upward pointing triangle.

Report Usage

The MAX attribute specifies printing the maximum value the STRING control's USE variable has contained so far. Unless the TALLY attribute is present, the result is calculated as follows:

- A MAX field in a DETAIL structure is evaluated each time the DETAIL structure containing the control PRINTs. This provides a "running" maximum value.
- A MAX field in a group FOOTER structure is evaluated each time any DETAIL structure in the BREAK structure containing the control PRINTs. This provides the maximum value of the variable in the group.
- A MAX field in a page FOOTER structure is evaluated each time any DETAIL structure in any BREAK structure PRINTs. This is the maximum value of the variable in the page (or report to date).
- A MAX field in a HEADER is meaningless, since no DETAIL structures will have been printed at the time the HEADER is printed.

The MAX value is reset only if the RESET or PAGE attribute is also specified.

Example:

```
!A Window with a maximize button:
```

```
Win2 WINDOW,MAX
```

```
END
```

```
CustRpt REPORT,AT(1000,1000,6500,9000),THOUS
```

```
Break1 BREAK(LocalVar),USE(?BreakOne)
```

```
Break2 BREAK(Pre:Key1),USE(?BreakTwo)
```

```
Detail DETAIL,AT(0,0,6500,1000),USE(?DetailOne)
```

```
STRING(@N$11.2),AT(6000,1500,500,500),USE(Pre:F1)
```

```
END
```

```
FOOTER,AT(0,0,6500,1000),USE(?BreakOneGroupFoot)
```

```
STRING('Group Maximum: '),AT(5500,500)
```

```
STRING(@N$11.2),AT(6000,500),USE(Pre:F1),MAX(LocalVar),RESET(Break2)
```

```
END
```

```
END
```

```
FOOTER,AT(0,0,6500,1000),USE(?BreakOneGroupFoot)
```

```
STRING('Grand Maximum: '),AT(5500,500)
```

```
STRING(@N$11.2),AT(6000,500),USE(LocalVar),MAX,TALLY(?BreakTwo)
```

```
END
```

```
END
```

```
END
```

See Also:

ICONIZE

ICON

MAXIMIZE

IMM

TALLY

RESET

PAGE

MAXIMIZE (set window open maximized)

MAXIMIZE

The **MAXIMIZE** attribute (PROP:MAXIMIZE) specifies the APPLICATION or WINDOW is opened maximized. When maximized, an APPLICATION or non-MDI WINDOW expands to occupy the full screen, and an MDI WINDOW expands to occupy the entire APPLICATION. Once expanded, the maximize control appears as an upward pointing triangle above a downward pointing triangle (in Windows 3.1).

With the SYSTEM built-in variable as the property assignment *target*, PROP:MAXIMIZE returns 1 if the Windows PIF setting for the application is set to open the application maximized.

Example:

```
!A Window with a maximize button, opened maximized:  
Win2 WINDOW,MAX,MAXIMIZE  
END
```

See Also:

MAX

IMM

MDI (set MDI child window)

MDI

The **MDI** attribute (PROP:MDI, read-only) specifies a WINDOW that acts as a "child" window to an APPLICATION. MDI windows are clipped to the APPLICATION frame--they display only in the APPLICATION's client area. MDI windows automatically move when the APPLICATION frame is moved, and are totally concealed by minimizing the APPLICATION. An MDI WINDOW cannot open until there is an active APPLICATION.

Modeless Windows

MDI "child" windows are modeless; the user may change to the top window of another execution thread, within the same application or any other application running in Windows, at any time. An MDI "child" window must not be on the same execution thread as the APPLICATION. Therefore, any MDI "child" window called directly from the APPLICATION must be in a separate procedure so the START procedure can be used to begin a new execution thread. Once started, multiple MDI "child" windows may be called in the new thread.

Application Modal Windows

A non-MDI WINDOW operates independently of any previously opened APPLICATION. It will, however, disable an APPLICATION if it or any of its MDI "child" windows are on the same execution thread as the non-MDI window. This makes a non-MDI window opened in an MDI thread an "application modal" window which effectively disables the application while the user has the window open (unless it is opened in its own separate execution thread). It does not, however, prevent the user from changing to another application running under Windows. An MDI window may not be opened on the same thread as an already open non-MDI window.

Example:

```
Win2 WINDOW,MDI      !An MDI child Window
      END
```

See Also:

MODAL

THREAD

MIN (set total minimum)**MIN**([*variable*])

MIN	Calculates the minimum value the STRING control's USE variable has contained so far.
<i>variable</i>	The label of a numeric variable to receive the intermediate values calculated for the MIN. This allows you to create totals on other totals. The value in the <i>variable</i> is internally updated by the print engine, so it is only useful for use within the REPORT structure.

The **MIN** attribute (PROP:MIN) specifies printing the minimum value the STRING control's USE variable has contained so far. Unless the TALLY attribute is present, the result is calculated as follows:

- A MIN field in a DETAIL structure is evaluated each time the DETAIL structure containing the control PRINTs. This provides a "running" minimum value.
- A MIN field in a group FOOTER structure is evaluated each time any DETAIL structure in the BREAK structure containing the control PRINTs. This provides the minimum value of the variable in the group.
- A MIN field in a page FOOTER structure is evaluated each time any DETAIL structure in any BREAK structure PRINTs. This is the minimum value of the variable in the page (or report to date).
- A MIN field in a HEADER is meaningless, since no DETAIL structures will have been printed at the time the HEADER is printed.

The MIN value is reset only if the RESET or PAGE attribute is also specified.

Example:

```

CustRpt REPORT,AT(1000,1000,6500,9000),THOUS
Break1  BREAK(LocalVar),USE(?BreakOne)
Break2  BREAK(Pre:Key1),USE(?BreakTwo)
Detail  DETAIL,AT(0,0,6500,1000),USE(?DetailOne)
        STRING(@N$11.2),AT(6000,1500,500,500),USE(Pre:F1)
        END
        FOOTER,AT(0,0,6500,1000),USE(?BreakOneGroupFoot)
        STRING('Group Minimum: '),AT(5500,500)
        STRING(@N$11.2),AT(6000,500),USE(Pre:F1),MIN(LocalVar),RESET(Break2)
        END
        END
        FOOTER,AT(0,0,6500,1000),USE(?BreakOneGroupFoot)
        STRING('Grand Minimum: '),AT(5500,500)
        STRING(@N$11.2),AT(6000,500),USE(LocalVar),MIN,TALLY(?BreakTwo)
        END
END;END

```

MODAL (set system modal window)

MODAL

MODAL has no effect for 32-bit applications, and has been deprecated in this release. The Microsoft Win32 API does not support system modal windows.

Application Modal Windows

A WINDOW without the MODAL attribute, may be either "application-modal" or "modeless." An application-modal window is a non-MDI window opened as the top window of an MDI execution thread. An application-modal window restricts the user from moving to another execution thread in the same application, but does not restrict them from changing to another Windows program.

Modeless Windows

A modeless window is an MDI "child" WINDOW (with the MDI attribute) without the MODAL attribute. From a modeless window, The top window on other execution threads may be selected by the mouse, keyboard, or menu commands. If so, the other window takes focus and becomes uppermost on the video display. Any window not on the top of its execution thread may not be selected to receive focus, even from a modeless window.

Example:

```
Win2 WINDOW,MODAL    !A system-modal Window
END
```

See Also:

MDI

THREAD

MSG (set status bar message)**MSG**(*text*)**MSG** Specifies *text* to display in the status bar.*text* A string constant containing the message to display in the status bar.The **MSG** attribute (PROP:MSG) specifies *text* to display in the first zone of the status bar.

On a control declaration, MSG specifies the *text* to display when the control has focus. If the control has non-persistent focus (has the SKIP attribute, or is placed in a TOOLBAR or a window with the TOOLBOX attribute) the *text* displays whenever the mouse cursor is positioned over the control.

On an APPLICATION or WINDOW structure, MSG specifies *text* to display in the first zone of the status bar when the control with focus has no MSG attribute of its own.

Example:

```

MainWin APPLICATION('My Application'),SYSTEM,MAX,ICON('MyIcon.ICO'),STATUS
  MENUBAR
    MENU('&File'),USE(?FileMenu)
    ITEM('&Open...'),USE(?OpenFile),MSG('Open a file')
    ITEM('&Close'),USE(?CloseFile),DISABLE,MSG('Close the open file')
    ITEM(),SEPARATOR
    ITEM('E&xit'),USE(?MainExit),MSG('Exit the program')
  END
END
END

WinOne WINDOW,AT(0,0,160,400),MSG('Enter Data')      !Default MSG to use
  COMBO(@S8),AT(120,120,20,20),USE(?C1),FROM(Q1:F2),MSG('Enter or Select')
  TEXT,AT(20,0,40,40),USE(E2)                       !Default MSG used
  ENTRY(@S8),AT(100,200,20,20),USE(E2)              !Default MSG used
  CHECK('&A'),AT(0,120,20,20),USE(?C7),MSG('On or Off')
  OPTION('Option 1'),USE(OptVar),MSG('Pick One or Two')
  RADIO('Radio 1'),AT(120,0,20,20),USE(?R1)
  RADIO('Radio 2'),AT(140,0,20,20),USE(?R2)
END
END

```

See Also:

STATUS

NOBAR (set no highlight bar)**NOBAR**

The **NOBAR** attribute (PROP:NOBAR) specifies the currently selected element in the LIST is only highlighted when the LIST control has focus.

NOCASE (case insensitive report BREAK)

NOCASE

The **NOCASE** attribute (PROP:NOCASE) of a BREAK structure in a REPORT declaration makes the comparison for detecting a changed value (indicating a group break) insensitive to the ASCII upper/lower case sorting convention. All characters in the break field and the saved comparison value are converted to upper case before the comparison. This case conversion has no effect on the case of the stored data. The NOCASE attribute has no effect on non-alphabetic characters.

Example:

```
Report REPORT
    BREAK(BreakVariable),NOCASE !Case insensitive group break
    HEADER
        STRING(@n4),USE(BreakVariable)
    END
Detail  DETAIL
        STRING(@n4),USE(SomeField)
    END
    END
END
```

See Also:

BREAK

NOMERGE (set merging behavior)**NOMERGE**

The **NOMERGE** attribute (PROP:NOMERGE) indicates that the MENUBAR or TOOLBAR on a WINDOW should not merge with the Global menu or toolbar.

The NOMERGE attribute on an APPLICATION's MENUBAR indicates that the menu is local and to be displayed only when no MDI "child" windows are open and that there is no Global menu. The NOMERGE attribute on an APPLICATION's TOOLBAR indicates that the tools are local and display only when no MDI "child" windows are open--there are no Global tools.

Without the NOMERGE attribute, an MDI WINDOW's menu and toolbar are automatically merged with the global menu and toolbar, and then displayed in the APPLICATION menu and toolbar. When NOMERGE is specified, the WINDOW's menu and toolbar overwrite the Global menu and toolbar. The menu and toolbar displayed when the WINDOW has focus are only the WINDOW's own menu and toolbar. However, they are still displayed on the APPLICATION.

A MENUBAR or TOOLBAR specified in a non-MDI WINDOW is never merged with the Global menu or toolbar--they appear in the WINDOW.

Examples:

```
!An MDI application frame window with local-only menu and toolbar:
MainWin APPLICATION('My Application'),SYSTEM,MAX,ICON('MyIcon.ICO'),STATUS
    MENUBAR,NOMERGE
        ITEM('E&xit'),USE(?MainExit)
    END
    TOOLBAR,NOMERGE
        BUTTON('Exit'),USE(?MainExitButton)
    END
END
```

```
!MDI window with its own menu and toolbar, overwriting the application's:
MDIChild WINDOW('Dialog Window'),MDI,SYSTEM,MAX,STATUS
  MENUBAR,NOMERGE
  MENU('Edit'),USE(?EditMenu)
  ITEM('Cu&t'),USE(?CutText),KEY(CTRLX),STD(STD:Cut)
  ITEM('Copy'),USE(?CopyText),KEY(CTRLC),STD(STD:Copy)
  ITEM('Paste'),USE(?PasteText),KEY(CTRLV),STD(STD:Paste)
  END
  END
  TOOLBAR,NOMERGE
  BUTTON('Cut'),USE(?CutButton),STD(STD:Cut)
  BUTTON('Copy'),USE(?CopyButton),STD(STD:Copy)
  BUTTON('Paste'),USE(?PasteButton),STD(STD:Paste)
  END
  TEXT,HVSCROLL,USE(Pre:Field),MSG('Enter some text here')
  BUTTON('&OK'),USE(?Exit),DEFAULT
  END
```

See Also:

MENUBAR

TOOLBAR

NOSHEET (set "floating" TABs)**NOSHEET**

The **NOSHEET** attribute (PROP:NOSHEET) on a SHEET control specifies the TABs display without a visible sheet to contain the controls. This creates a "floating tab" effect.

OPEN (open OLE control object from file)**OPEN**(*object*)

OPEN	Specifies opening a saved object for the OLE control from an OLE Compound Storage file.
<i>object</i>	A string constant containing the name of the OLE Compound Storage file and the object within it to open.

The **OPEN** attribute (PROP:OPEN, write-only) specifies opening a saved *object* for the OLE control from an OLE Compound Storage file. When the object is opened, the saved version of the container properties are reloaded, so properties do not need to be specified on an object opened. The *object* parameter syntax must take the form: *Filename\ObjectName*.

Example:

```
WinOne WINDOW,AT(0,0,200,200)
      OLE,AT(10,10,160,100),USE(?OLEObject),OPEN('SavFile.OLE\!MyObject')
      MENUBAR
      MENU('&Clarion App')
      ITEM('&Deactivate Object'),USE(?DeactOLE)
      END
      END
      END
      END
```

PAGE (set page total reset)**PAGE**

The **PAGE** attribute (PROP:PAGE) specifies the CNT, SUM, AVE, MIN, or MAX is reset to zero (0) when page break occurs.

PAGEAFTER (set page break after)

PAGEAFTER([*newpage*])

PAGEAFTER Specifies the structure is printed, then initiates page overflow.

newpage An integer constant or constant expression that specifies the page number to print on the next page (PROP:PageAfterNum, equivalent to {PROP:PageAfter,2}). If zero (0) or omitted, forced page overflow does not occur. If negative one (-1), the current page number increments during page overflow.

The **PAGEAFTER** attribute (PROP:PAGEAFTER) specifies that the DETAIL, or group HEADER or FOOTER structure (contained within a BREAK structure), initiates page overflow after it is printed. This means that the print structure on which the PAGEAFTER attribute is present is printed, followed by the page FOOTER, and then the FORM and page HEADER.

The *newpage* parameter, if present, resets automatic page numbering at the number specified.

Example:

```
CustRpt  REPORT
          HEADER
          !structure elements
          END
Break1   BREAK(SomeVariable)
          HEADER
          !structure elements
          END
CustDetail DETAIL
          !structure elements
          END
          FOOTER,PAGEAFTER(-1)      !Group Footer, initiates page overflow
          !structure elements
          END
          END
          FOOTER
          !structure elements
          END
          END
```

PAGEBEFORE (set page break first)

PAGEBEFORE([*newpage*])

PAGEBEFORE Specifies the structure is printed on a new page, after page overflow.

newpage An integer constant or constant expression that specifies the page number to print on the new page (PROP:PageBeforeNum, equivalent to {PROP:PageBefore,2}). If zero (0) or omitted, forced page overflow does not occur. If negative one (-1) , the current page number increments during page overflow.

The **PAGEBEFORE** attribute (PROP:PAGEBEFORE) specifies that the DETAIL, or group HEADER or FOOTER structure (contained within a BREAK structure), is printed on a new page, after page overflow. This means that first, the page FOOTER is printed, then the FORM and page HEADER. The print structure on which the PAGEBEFORE attribute is present is printed only after these page overflow actions are complete.

The *newpage* parameter, if present, resets automatic page numbering at the number specified.

Example:

```
CustRpt REPORT
      HEADER
      !structure elements
      END
Break1   BREAK(SomeVariable)
      HEADER,PAGEBEFORE(-1)      !Group Header, initiates page overflow
      !structure elements
      END
CustDetail DETAIL
      !structure elements
      END
      FOOTER
      !structure elements
      END
      END
      FOOTER
      !structure elements
      END
      END
```

PAGENO (set page number print)**PAGENO**

The **PAGENO** attribute (PROP:PAGENO) specifies the STRING control prints the current page number.

PALETTE (set number of hardware colors)**PALETTE**(*colors*)

PALETTE	Specifies the number of hardware colors displayed in the window.
<i>colors</i>	An integer constant specifying the number of hardware colors displayed in the window.

The **PALETTE** attribute (PROP:PALETTE) on an APPLICATION or WINDOW structure specifies how many colors in the hardware palette you want this window to use when it is the foreground window. This is only applicable in hardware modes where a palette is in use and spare colors (not reserved by the system) are available - in practice this means 256 color mode. This enforces a particular set of colors for the graphics. 24-bit color (16.7M) does not use a hardware palette. Values of PALETTE above 256 are not recommended.

The value returned by PROP:PALETTE is the number of colors actually assigned (not necessarily the number asked for). Since the system normally reserves 20 in 256-color mode, setting PROP:PALETTE = 256 then immediately retrieving its value will usually result in a returned value of 236.

Example:

```
WinOne WINDOW,AT(0,0,160,400),PALETTE(26)      !Display 26-color
        IMAGE,AT(120,120,20,20),USE(ImageField)
        END
```

See Also:

IMAGE

PAPER (set report paper size)

PAPER([*type*] [,*width*] [,*height*])

PAPER	Defines the paper size for the report.
<i>type</i>	An integer constant or EQUATE that specifies a standard Windows paper size. EQUATES for these are contained in the PRNPROP.CLW file.
<i>width</i>	An integer constant or constant expression that specifies the width of the paper (PROPPRINT:paperwidth, equivalent to {PROPPRINT:PAPER,2})
<i>height</i>	An integer constant or constant expression that specifies the height of the paper (PROPPRINT:paperheight, equivalent to {PROPPRINT:PAPER,3}).

The **PAPER** attribute (PROPPRINT:PAPER) on a REPORT structure defines the paper size for the report. The *width* and *height* parameters are only required when PAPER:User is selected as the *type*. Not all printers support all paper sizes.

The values contained in the *width*, and *height* parameters default to dialog units unless the THOUS, MM, or POINTS attribute is also present. Dialog units are defined as one-quarter the average character width by one-eighth the average character height. The size of a dialog unit is dependent upon the size of the default font for the report. This measurement is based on the font specified in the FONT attribute of the report, or the printer's default font.

Example:

```
CustRpt1 REPORT,AT(1000,1000,6500,9000),THOUS,PAPER(PAPER:Custom,8500,7000)
!print on 8.5" x 7" paper
!report declarations
END
```

```
CustRpt2 REPORT,AT(72,72,468,648),POINTS,PAPER(PAPER:A4)
!print on A4 size paper
!report declarations
END
```

PASSWORD (set data non-display)**PASSWORD**

The **PASSWORD** attribute (PROP:PASSWORD) specifies non-display of the data entered in the ENTRY control. When the user types in data, asterisks are displayed on screen for each character entered. The Windows standard Cut and Copy features are disabled when the PASSWORD attribute is active.


```

LOOP
  NEXT(SomeFile)
  IF ERRORCODE() THEN BREAK.
  PRINT(DetailOne)
END
ENDPAGE(Report)
OPEN(ViewReport)                !Open report preview window
GET(WMFQue,NextEntry)           !Get first queue entry
?ImageField{PROP:text} = WMFQue.PageImage !Load first report page
ACCEPT
CASE ACCEPTED()
OF ?NextPage
  NextEntry += 1                !Increment entry counter
  IF NextEntry > RECORDS(WMFQue) !Check for end of report
    CYCLE
  END
  GET(WMFQue,NextEntry)         !Get next queue entry
  ?ImageField{PROP:text} = WMFQue.PageImage !Load next report page
  DISPLAY                       !and display it
OF ?PrintReport
  Report{PROP:flushpreview} = TRUE !Flush files to printer
  BREAK                         !and exit procedure
OF ?ExitReport
  BREAK                         !Exit procedure
END
END
CLOSE(ViewReport)              !Close window
FREE(WMFQue)                   !Free the queue memory
CLOSE(Report)                  !Close report (deletes all .WMF files)
RETURN                          !and return to caller

```

RANGE (set range limits)*RANGE(lower,upper)*

RANGE	Specifies the valid range of data values the user may select in a SPIN control, or the range of values displayed in a PROGRESS control.
<i>lower</i>	A numeric constant that specifies the lower inclusive limit of valid data (PROP:RangeLow, equivalent to {PROP:Range,1}).
<i>upper</i>	A numeric constant that specifies the upper inclusive limit of valid data (PROP:RangeHigh, equivalent to {PROP:Range,2}).

The **RANGE** attribute (PROP:RANGE) specifies the valid range of data values the user may select in a SPIN control. RANGE also defines the range of values that are displayed in a PROGRESS control. This attribute works in conjunction with the STEP attribute on SPIN controls. On a SPIN control, the STEP attribute provides the user with the valid choices within the range.

PROP:RangeHigh returns "+Infinity" if no RANGE is set. PROP:RangeLow returns "-Infinity" if no RANGE is set.

PROP:RangeLow and PROP:RangeHigh are also applicable for LIST and COMBO controls with vertical scrollbars (Read Only). PROP:RangeLow returns the lower limit of possible PROP:VScrollPos values, and PROP:RangeHigh returns upper limit. The limits or boundaries mentioned in the description of PROP:VScrollPos are correct only if number of records in the list box source is less or equal to number of visible rows (i.e., for page loaded browses).

Example:

```
WinOne WINDOW,AT(0,0,160,400)
    SPIN(@N4.2),AT(280,0,20,20),USE(SpinVar1),RANGE(.05,9.95),STEP(.05)
    SPIN(@n3),AT(280,0,20,20),USE(SpinVar2),RANGE(5,995),STEP(5)
END
```

READONLY (set display-only)**READONLY**

The **READONLY** attribute (PROP:READONLY) specifies a display-only COMBO, ENTRY, SPIN or TEXT control. The control may receive input focus with the mouse, but may not enter data. If the user attempts to change the displayed value, a beep warns the user that data entry is not allowed.

REPEAT (set repeat button rate)**REPEAT**(*time*)**REPEAT** Specifies the rate of event generation.*time* An integer constant containing the rate to set, in hundredths of a second.

The **REPEAT** attribute (PROP:REPEAT) specifies rate of event generation for automatically repeating buttons. For a **BUTTON** control with the **IMM** attribute, this is the generation rate for **EVENT:Accepted**. For a **SPIN** control, this is the generation rate for **EVENT:NewSelection** generated by the spin buttons.

Assigning a zero (0) to PROP:REPEAT resets the default setting, any other value sets the repeat rate for the control.

Example:

```
MDIChild WINDOW( 'Child One' ), AT(0,0,320,200), MDI, MAX, HVSCROLL
    BUTTON( 'Press Me' ), AT(10,10,40,20), USE(?PressMe), IMM, REPEAT(100)    !1/second
    SPIN(@n3), AT(60,10,40,10), USE(SpinVar), RANGE(0,999), REPEAT(100)    !1/second
END

CODE
OPEN(MDIChild)
?PressMe{PROP:Delay} = 50    !Set delay to 1/2 second
?SpinVar{PROP:Delay} = 50    !Set delay to 1/2 second
?PressMe{PROP:Repeat} = 5    !Reset repeat to 5 hundredths of a second
?SpinVar{PROP:Repeat} = 5    !Reset repeat to 5 hundredths of a second
```

See Also:

IMM**DELAY**

REQ (set required entry)**REQ**

The **REQ** attribute (PROP:REQ) specifies an ENTRY or TEXT control that may not be left blank or zero. The REQ attribute on an ENTRY or TEXT control is not checked until a BUTTON with the REQ attribute is pressed, or the INCOMPLETE() procedure is called.

When a BUTTON with the REQ attribute is pressed, or the INCOMPLETE() procedure is called, all ENTRY and TEXT controls with the REQ attribute are checked to ensure they contain data. The first control encountered in this check that does not contain data immediately receives input focus.

RESET (set total reset)*RESET(breaklevel)***RESET** Resets the CNT, SUM, AVE, MIN, or MAX to zero (0).*breaklevel* The label of a BREAK structure.

The **RESET** attribute (PROP:RESET) specifies the group break at which the CNT, SUM, AVE, MIN, or MAX is reset to zero (0). PROP:RESET returns zero (0) if not present, else it returns the breaklevel nesting depth.

Example:

```

CustRpt REPORT,AT(1000,1000,6500,9000),THOUS
Break1  BREAK(Pre:Key1)
        HEADER,AT(0,0,6500,1000)
          STRING('Group Head'),AT(3000,500,1500,500),FONT('Arial',18)
        END
Detail  DETAIL,AT(0,0,6500,1000)
        STRING(@N$11.2),AT(6000,1500,500,500),USE(Pre:F1)
        END
        FOOTER,AT(0,0,6500,1000)
          STRING('Group Total:'),AT(5500,500,1500,500)
          STRING(@N$11.2),AT(6000,500,500,500),USE(Pre:F1),SUM,RESET(Break1)
        END
        END
        END

```

RESIZE (set variable height TEXT control)

RESIZE

The **RESIZE** attribute (PROP:RESIZE) specifies height of the TEXT control varies according to the amount of data to print in it, up to the maximum height specified by the control's AT attribute.

The *height* parameter in the AT attribute of the DETAIL, HEADER, or FOOTER structure containing the TEXT control must not be set (let it default) for the RESIZE attribute to have any effect.

Example:

```
CustRpt REPORT,AT(1000,1000,6500,9000),THOUS
Detail  DETAIL,AT(0,0,6500,)                !Detail height defaults
        STRING(@N$11.2),AT(500,500,500,) ,USE(Pre:F1)
        TEXT,AT(500,1000,500,5000),USE(Pre:Mem01),RESIZE !Print height up to 5"
        END
        END
```

RIGHT (set MENU position)**RIGHT**

The **RIGHT** attribute (PROP:RIGHT) specifies the MENU is placed at the right end of the action bar.

ROUND (set round-cornered BOX)**ROUND**

The **ROUND** attribute (PROP:ROUND) specifies a BOX control with rounded corners.

RTF (declare TEXT control as RichEdit)

RTF(*savetarget*)

RTF	Declare a TEXT field with RTF attributes.
<i>savetarget</i>	A byte or EQUATE that specifies that the TEXT control will be saved to a Field (TEXT:FIELD or 0) or File (TEXT:FILE or 1)

The RTF attribute declares a TEXT control as one with Rich Text Control attributes. The necessary code is generated and used to manipulate the text within the control as Rich Text.

Clarion's implementation of Rich Text allows the rich text field to be saved to a field or a separate text file. The minimum size of a target field is 256 characters. This is needed in order to maintain the rich text header information. If the field contains a lot of formatting a larger field size will be needed.

The components of a rich text control include a Toolbar, Format Bar, and Ruler. These components of the rich text control are optional. They may be shown or hidden at runtime. They default to showing at runtime. The bars may not be modified at design time in the window formatter. When the control is placed on the window, it is a basic TEXT control.

SCROLL (set scrolling control)

SCROLL

The **SCROLL** attribute (PROP:SCROLL) specifies a control that moves with the window when the WINDOW scrolls. This allows "virtual" windows larger than the physical video display.

The presence of the SCROLL attribute means that the control stays fixed at a position in the window relative to the top left corner of the virtual window, whether that position is currently in view or not. This means that the control appears to move as the window scrolls.

If the SCROLL attribute is omitted, the control stays fixed at a position in the window relative to the top left corner of the currently visible portion of the window. This means that the control appears to stay in the same position on screen while the rest of the window scrolls, which is useful for controls which should stay visible to the user at all times (such as Ok or Cancel buttons).

Mixing controls with and without the SCROLL attribute on the same WINDOW can result in multiple controls appearing to occupy the same screen position. This occurs because the controls with SCROLL move and the controls without SCROLL do not. This condition is temporary and scrolling the window further will correct the situation. The situation can be avoided entirely by careful placement of controls in the window. For example, you can place all controls without SCROLL at the bottom of the window then place all controls with SCROLL above them extending to the right and left. This would create a window that is designed to scroll horizontally (the WINDOW should have the HSCROLL attribute and not the VSCROLL or HVSCROLL attributes).

SEPARATOR (set separator line ITEM)**SEPARATOR**

The **SEPARATOR** attribute specifies an ITEM in a MENU that displays a horizontal line to group ITEMS within the MENU. The USE attribute may be specified for the ITEM.

Example:

```
ITEM,SEPARATOR,USE(?FileSeprator1)
```

SINGLE (set TEXT for single line entry)**SINGLE**

The **SINGLE** attribute (PROP:SINGLE) specifies the control is only for single line data entry. This is specifically to allow use of TEXT controls instead of ENTRY for data entry in languages that write from right to left (such as Hebrew or Arabic).

SKIP (set Tab key skip or conditional print control)**SKIP**

The **SKIP** attribute (PROP:SKIP) on a window control specifies user access to the control is only with the mouse or an accelerator key, not the TAB key. Data entry controls receive input focus only during data entry and the control does not retain focus, while non-data entry controls do not receive or retain input focus (the same behavior demonstrated by controls in a toolbar or toolbox).

When the mouse is over a control with the SKIP attribute, the control's MSG attribute displays in the status bar.

The SKIP attribute on a report control specifies the STRING or TEXT control prints only if its USE variable contains data. If the USE variable does not contain data, the STRING or TEXT control does not print and all controls following in the band "move up" to fill in the space. This is most useful for label printing to prevent extra blank lines in addresses.

Example:

```
CustRpt REPORT,AT(1000,1000,6000,9000),THOUS
Detail  DETAIL,AT(0,0,2000,1000)           !Fixed height detail
        STRING(@s35),AT(250,250,500,),USE(Pre:Name)
        STRING(@s35),AT(250,250,500,),USE(Pre:Address1)
        STRING(@s35),AT(250,250,500,),USE(Pre:Address2),SKIP !don't print if blank
        STRING(@s35),AT(250,250,500,),USE(CityStateZip)    ! and move this up
        END
        END
```

SMOOTH (set smooth progress bar increments)**SMOOTH**

The **SMOOTH** attribute (PROP:SMOOTH) activates a smooth incremental display of the progress control instead of the standard "block" format. Valid only for a PROGRESS control.

SPREAD (set evenly spaced TAB controls)**SPREAD**

The **SPREAD** attribute (PROP:SPREAD) specifies a SHEET's TAB controls are evenly spaced.

STATUS (set status bar)

STATUS([*widths*])

STATUS	Specifies the presence of a status bar.
<i>widths</i>	A list of integer constants (separated by commas) specifying the size of each zone in the status bar. If omitted, the status bar has one zone the width of the window.

The **STATUS** attribute (PROP:STATUS) specifies the presence of a status bar at the base of the APPLICATION or WINDOW. The status bar of an MDI WINDOW is always displayed at the bottom of the APPLICATION. A WINDOW without the MDI attribute displays its status bar at the base of the WINDOW. If the STATUS attribute is not present on the APPLICATION or WINDOW, there is no status bar.

The status bar may be divided into multiple zones specified by the *widths* parameters. The size of each zone is specified in dialog units. A negative value indicates the zone is expandable, but has a minimum width indicated by the parameter's absolute value. If no *widths* parameters are specified, a single expanding zone with no minimum width is created, which is equivalent to a STATUS(-1).

PROP:STATUS contains the *widths* of each status bar section in separate array elements. A zero (0) value is required in the last element to terminate the array.

The first zone of the status bar is always used to display MSG attributes. The MSG attribute string is displayed in the status bar as long as its control field still has input focus. A control or menu item without a MSG attribute causes the status bar to revert to its former state (either blank or displaying the text previously displayed in the zone).

Text may be placed in, or retrieved from, any zone of the status bar using runtime property assignment to PROP:StatusText. PROP:StatusText is an array containing the text of each section of the status bar. A zero (0) value is required in the last element to terminate the array. The text remains present until replaced.

Example:

```
!An APPLICATION with a one-zone status bar:
```

```
MainWin APPLICATION,STATUS  
      END
```

```
!A WINDOW with a two-zone status bar:
```

```
Win1 WINDOW,STATUS(160,160)  
      END
```

```
CODE
```

```
OPEN(Win1)
```

```
Win1{PROP:STATUS,3} = 160
```

```
!Add a status bar zone
```

```
Win1{PROP:STATUS,4} = 0
```

```
!and terminate the array
```

```
Win1{PROP:StatusText,3} = 'Hello Zone 3' !Put text in the new zone
```

See Also:

MSG

STD (set standard behavior)*STD(behavior)*

STD	Specifies standard Windows <i>behavior</i> .
<i>behavior</i>	An integer constant or EQUATE specifying the identifier of a standard windows behavior.

The **STD** attribute (PROP:STD) specifies the control activates some standard Windows action. This action is automatically executed by the runtime library and does not generate any events (that is, the control does not receive an EVENT:Accepted).

EQUATE statements for the standard Windows actions are contained in the EQUATES.CLW file. The following list is a representative sample of these (see EQUATES.CLW for the complete list):

STD:WindowList	List of open MDI windows
STD:TileWindow	Tile Windows
STD:CascadeWindow	Cascade Windows
STD:ArrangeIcons	Arrange Icons
STD:HelpIndex	Help Contents
STD:HelpSearch	Help Search dialog

Example:

```
MDIChild WINDOW('Child One'),MDI,SYSTEM,MAX
  MENUBAR
    MENU('Edit'),USE(?EditMenu)
      ITEM('Undo'),USE(?UndoText),KEY(CtrlZ),STD(STD:Undo)
      ITEM('Cu&t'),USE(?CutText),KEY(CtrlX),STD(STD:Cut)
      ITEM('Copy'),USE(?CopyText),KEY(CtrlC),STD(STD:Copy)
      ITEM('Paste'),USE(?PasteText),KEY(CtrlV),STD(STD:Paste)
    END
  END
  TOOLBAR
    BUTTON('Cut'),USE(?CutButton),ICON(ICON:Cut),STD(STD:Cut)
    BUTTON('Copy'),USE(?CopyButton),ICON(ICON:Copy),STD(STD:Copy)
    BUTTON('Paste'),USE(?PasteButton),ICON(ICON:Paste),STD(STD:Paste)
  END
END
```

STEP (set SPIN increment)**STEP**(*count*)

STEP Specifies a SPIN control RANGE attribute's increment/decrement value.*count* A numeric constant specifying the amount to increment or decrement.

The **STEP** attribute (PROP:STEP) specifies the amount by which a SPIN control's value is incremented or decremented within its valid RANGE. The default STEP value is 1.0.

Example:

```
WinOne WINDOW,AT(0,0,160,400)
    SPIN(@N4.2),AT(280,0,20,20),USE(SpinVar1),RANGE(.05,9.95),STEP(.05)
    SPIN(@N3),AT(280,0,20,20),USE(SpinVar2),RANGE(5,995),STEP(5)
    SPIN(@T3),AT(280,0,20,20),USE(SpinVar3),RANGE(1,8640000),STEP(6000)
END
```

STRETCH (set OLE object stretching)**STRETCH**

The **STRETCH** attribute (PROP:STRETCH, write-only) specifies the OLE object stretches to completely fill the size specified by the OLE container control's AT attribute. This attribute does not preserve the object's aspect ratio.

SUM (set total)

SUM([*variable*])

SUM	Calculates the the sum of the values contained in the STRING control's USE variable.
<i>variable</i>	The label of a numeric variable to receive the intermediate values calculated for the SUM. This allows you to create totals on other totals. The value in the <i>variable</i> is internally updated by the print engine, so it is only useful for use within the REPORT structure.

The **SUM** attribute (PROP:SUM) specifies printing the sum of the values contained in the STRING control's USE variable. Unless the TALLY attribute is present, the result is calculated as follows:

- A SUM field in a DETAIL structure increments each time the DETAIL structure containing the control PRINTs. This provides a "running" total.
- A SUM field in a group FOOTER structure increments each time any DETAIL structure in the BREAK structure containing the control PRINTs. This provides the sum of the value contained in the variable in the group.
- A SUM field in a page FOOTER structure increments each time any DETAIL structure in any BREAK structure PRINTs. This is the sum of the values contained in the variable in the page.
- A SUM field in a HEADER is meaningless, since no DETAIL structures will have been printed at the time the HEADER is printed.

The SUM value is reset only if the RESET or PAGE attribute is also specified.

Example:

```
CustRpt REPORT,AT(1000,1000,6500,9000),THOUS
Break1 BREAK(LocalVar),USE(?BreakOne)
Break2 BREAK(Pre:Key1),USE(?BreakTwo)
Detail  DETAIL,AT(0,0,6500,1000),USE(?DetailOne)
        STRING(@N$11.2),AT(6000,1500,500,500),USE(Pre:F1)
        END
        FOOTER,AT(0,0,6500,1000),USE(?BreakOneGroupFoot)
        STRING('Group Total:'),AT(5500,500)
        STRING(@N$11.2),AT(6000,500),USE(Pre:F1),SUM(LocalVar),RESET(Break2)
        END
        END
        FOOTER,AT(0,0,6500,1000),USE(?BreakOneGroupFoot)
        STRING('Grand Total:'),AT(5500,500)
        STRING(@N$11.2),AT(6000,500),USE(LocalVar),SUM,TALLY(?BreakTwo)
        END
        END
        END
```

SYSTEM (set system menu)**SYSTEM**

The **SYSTEM** attribute (PROP:SYSTEM) specifies the presence of a Windows system menu (also called the control menu) on the APPLICATION or WINDOW. This menu contains standard Windows menu selections, such as: Close, Minimize, Maximize (the window), and Switch To (another window). The actual selections available on a given window depend upon the attributes set for that window.

Example:

!An APPLICATION with a system menu:

```
MainWin APPLICATION,SYSTEM  
END
```

!A WINDOW with a system menu:

```
Win1 WINDOW,SYSTEM  
END
```

TALLY (set total calculation times)**TALLY**(*points*)

TALLY	Specifies when to calculate an AVE, CNT, MAX, MIN, or SUM.
<i>points</i>	A comma delimited list of the labels of the DETAIL and/or BREAK structures on which to calculate the total.

The **TALLY** attribute (PROP:TALLY) specifies when to calculate an AVE, CNT, MAX, MIN, or SUM. The appropriate total is calculated each time any of the DETAIL structures named in the *points* list prints, or in the case of a BREAK structure, when the group break occurs.

Example:

```

CustRpt REPORT,AT(1000,1000,6500,9000),THOUS
Break1  BREAK(LocalVar),USE(?BreakOne)
Break2  BREAK(Pre:Key1),USE(?BreakTwo)
        HEADER,AT(0,0,6500,1000),USE(?GroupHead)
        STRING('Group Head'),AT(3000,500,1500,500),FONT('Arial',18)
        END
Detail  DETAIL,AT(0,0,6500,1000),USE(?DetailOne)
        STRING(@N$11.2),AT(6000,1500,500,500),USE(Pre:F1)
        END
        END
        FOOTER,AT(0,0,6500,1000),USE(?BreakOneGroupFoot)
        STRING('Group Total:'),AT(5500,500,1500,500)
        STRING(@N$11.2),AT(6000,500,500,500),USE(Pre:F1),CNT,TALLY(Break2)
        END
        END
        END
CODE
OPENCustRpt)
CustRpt$?Pre:F1{PROP:Tally} = ?BreakOne !Change the TALLY to Break1

```

THOUS, MM, POINTS (set report coordinate measure)**THOUS****MM****POINTS**

The **THOUS**, **MM**, and **POINTS** attributes specify the coordinate measures used to position controls on the REPORT.

THOUS (PROP:THOUS) specifies thousandths of an inch, **MM** (PROP:MM) specifies millimeters, and **POINTS** (PROP:POINTS) specifies points (there are seventy-two points per inch, both vertically and horizontally).

If all these attributes are omitted, the measurements default to dialog units. Dialog units are defined as one-quarter the average character width by one-eighth the average character height. The size of a dialog unit is dependent upon the size of the default font for the report. This measurement is based on the font specified in the FONT attribute of the REPORT, or the system default font specified by Windows.

TILED (set tiled image)**TILED**

The **TILED** attribute (PROP:TILED) indicates that the image displayed in the IMAGE control, or the window or toolbar's background image (specified in the WALLPAPER attribute) displays at its default size and is tiled to fill the entire window, toolbar, or area specified by the IMAGE's AT attribute.

Example:

```
WinOne WINDOW,AT(,,380,200),MDI
    IMAGE('MyWall.GIF'),AT(0,0,380,200),TILED
END
MDIChild WINDOW('Child One'),MDI,SYSTEM,MAX
    MENUBAR
    MENU('Edit'),USE(?EditMenu)
    ITEM('Undo'),USE(?UndoText),KEY(CtrlZ),STD(STD:Undo)
    ITEM('Cu&t'),USE(?CutText),KEY(CtrlX),STD(STD:Cut)
    ITEM('Copy'),USE(?CopyText),KEY(CtrlC),STD(STD:Copy)
    ITEM('Paste'),USE(?PasteText),KEY(CtrlV),STD(STD:Paste)
    END
    END
    TOOLBAR,USE(?Toolbar),WALLPAPER('MyWall.GIF'),TILED
    BUTTON('Cut'),USE(?CutButton),ICON(ICON:Cut),STD(STD:Cut),FLAT
    BUTTON('Copy'),USE(?CopyButton),ICON(ICON:Copy),STD(STD:Copy),FLAT
    BUTTON('Paste'),USE(?PasteButton),ICON(ICON:Paste),STD(STD:Paste),FLAT
    END
    END
WinOne WINDOW,AT(,,380,200),MDI,WALLPAPER('MyWall.GIF'),TILED
    END
```

See Also:

CENTERED

WALLPAPER

TIMER (set periodic event)**TIMER**(*period*)

TIMER	Specifies a periodic event.
<i>period</i>	An integer constant or constant expression specifying the interval between timed events, in hundredths of a second. The maximum <i>period</i> you can specify is 6553 (a Windows limitation, approximately 65 seconds). If zero (0), no timed events generate.

The **TIMER** attribute (PROP:TIMER) specifies generation of a periodic field-independent event (EVENT:Timer) whenever the time *period* passes. The FOCUS() procedure returns the number of the control that currently has focus at the time of the event.

If the window that has the TIMER attribute does not have focus when EVENT:Timer occurs, the window that does have focus first receives EVENT:Suspend before the window with the TIMER attribute receives EVENT:Timer. After an EVENT:Suspend occurs in the window with focus, EVENT:Resume is generated before any other events are generated for that window, and the EVENT:Resume does not generate until there is another event to process in that window (the window is suspended and timer events continue processing until there is some activity to process in window with focus).

Example:

```
RunClock PROCEDURE
ShowTime LONG
```

```
!A WINDOW with a timed event occurring every second:
Win1 WINDOW,TIMER(100)
    STRING(@T4),USE(ShowTime)
    END
CODE
OPEN(Win1)
ShowTime = CLOCK()
ACCEPT
CASE EVENT()
OF EVENT:Timer
    ShowTime = CLOCK()
    DISPLAY
    END
END
CLOSE(Win1)
```

TIP (set "balloon help" text)TIP(*string*)

TIP	Specifies the text to display when the mouse cursor pauses over the control.
<i>string</i>	A string constant or expression that specifies the text to display and how the text is formatted (single or multi-line TIP).

The **TIP** attribute (PROP:Tip) on a control specifies the text to display in a "balloon help" box when the mouse cursor pauses over the control. Although there is no specific limit on the number of characters, the *string* (or any single line of a multi-line TIP) should not be longer than can be displayed on the screen.

Multi-line tip conventions are as follows for both the TIP Attribute, TIP(), or the TIP Property,{PROP:Tip}=:

All of the following will produce a two line TIP.

```
'Line1'& CHR(10)&'Line2' !CHR(10) = Line Feed
'Line1<13,10>Line2'      !<13,10> = Carriage Return, Line Feed
'Line1<10>Line2'         !<10> = Line Feed
```

Although it is valid on any control that can gain focus for user input, this attribute is most commonly used on BUTTON controls with the ICON attribute that are placed on the TOOLBAR. This allows the user to quickly determine the control's purpose without accessing the on-line Help system.

Automatic TIP attribute display can be disabled for any single control or window by setting the PROP:NoTips undeclared property to one (1). It can be disabled for an entire application by setting the PROP:NoTips for the built-in variable SYSTEM to one (1).

The amount of tool tip delay is an operating system setting under the user's control.

Example:

```
Win WINDOW,AT(0,0,160,400)
  TOOLBAR
    BUTTON('E&xit'),USE(?MainExitButton),ICON(ICON:hand),TIP('Exit Window')
    !Multi-line TIP:
    BUTTON('&Open'),USE(?OpenButton),ICON(ICON:Open),TIP('Open'<10>'a File')
  END
COMBO(@S8),AT(120,120,20,20),USE(?C1),FROM(Q1:F2)
ENTRY(@S8),AT(100,200,20,20),USE(E2)
END
```

TOGETHER (set to print on a single page)**TOGETHER**

The **TOGETHER** attribute (PROP:Together) specifies that the DETAIL, or group HEADER or FOOTER structure (contained within a BREAK structure), is to only print on a single page, rather than being automatically split if there is not sufficient room for it. The normal use is for structures with multiple lines that need to be printed together, as in mailing labels.

Example:

```
CustRpt   REPORT
TitlePage  DETAIL !Title page detail structure
           !structure elements
           END
CustDetail DETAIL,TOGETHER !Print all detail elements together
           !structure elements
           END
           FOOTER
           !structure elements
           END
           END
```

TOOLBOX (set toolbox window behavior)**TOOLBOX**

The **TOOLBOX** attribute (PROP:TOOLBOX) specifies a WINDOW that is "always on top" and may be docked if the DOCK attribute is also present. Neither the WINDOW nor its controls retain input focus. This creates control behavior as if all the controls in the WINDOW had the SKIP attribute.

Normally, a WINDOW with the TOOLBOX attribute executes in its own execution thread to provide a set of tools to the window with input focus. The MSG attributes of the controls in the window appear in the status bar when the mouse cursor is positioned over the control.

Example:

```

PROGRAM
MainWin APPLICATION('My Application')
  MENUBAR
  MENU('File'),USE(?FileMenu)
  ITEM('E&xit'),USE(?MainExit),LAST
  END
  MENU('Edit'),USE(?EditMenu)
  ITEM('Use Tools'),USE(?UseTools)
  END
  END
  END
  END

Pre:Field      STRING(400)
UseToolsThread BYTE
ToolsThread    BYTE

CODE
OPEN(MainWin)
ACCEPT
CASE ACCEPTED()
OF ?MainExit
  BREAK
OF ?UseTools
  UseToolsThread = START(UseTools)
END
END

UseTools PROCEDURE                                !A procedure that uses a toolbox
MDIChild WINDOW('Use Tools Window'),MDI
  TEXT,HVSCROLL,USE(Pre:Field)
  BUTTON('&OK'),USE(?Exit),DEFAULT
END

```

```

CODE
OPEN(MDICHild)                !Open the window
DISPLAY                       !and display it
ToolsThread = START(Tools)    !Pop up the toolbox
ACCEPT
CASE EVENT()                  !Check for user-defined events
OF 401h                       !posted by toolbox controls
  Pre:Field += ' ' & FORMAT(TODAY(),@D1) !append date to end of field
OF 402h                       !append time to end of field
  Pre:Field += ' ' & FORMAT(CLOCK(),@T1)
END
CASE ACCEPTED()
OF ?Exit
  POST(400h,,ToolsThread)      !Signal to close tools window
  BREAK
END
END
CLOSE(MDICHild)

Tools PROCEDURE !The toolbox procedure
Win1 WINDOW('Tools'),TOOLBOX
  BUTTON('Date'),USE(?Button1)
  BUTTON('Time'),USE(?Button2)
END

CODE
OPEN(Win1)
ACCEPT
IF EVENT() = 400h THEN BREAK.  !Check for close window signal
CASE ACCEPTED()
OF ?Button1
  POST(401h,,UseToolsThread)  !Post datestamp signal
OF ?Button2
  POST(402h,,UseToolsThread)  !Post timestamp signal
END
END
CLOSE(Win1)

```

See Also:

DOCK

TRN (set transparent control)

TRN

The **TRN** attribute (PROP:TRN) on a control specifies the characters print or display transparently, without obliterating the background over which the control is placed. Only the dots or pixels required to create each character are printed or displayed. This allows you to place the control directly on top of an **IMAGE** without destroying the background picture.

Example:

```
CustRpt REPORT,AT(1000,1000,6500,9000),THOUS
      FORM,AT(0,0,6500,9000)
      IMAGE('PIC.BMP'),USE(?I1)AT(0,0,6500,9000)      !Full page image
      !Transparent string on the image:
      STRING('String Constant'),AT(10,0,20,20),USE(?S1),TRN
      END
      END

WinOne WINDOW,AT(0,0,160,400)
      IMAGE('PIC.BMP'),USE(?I1),FULL                  !Full window image
      !Transparent string on image
      STRING('String Constant'),AT(10,0,20,20),USE(?S1),TRN
      END
```

UP, DOWN (set TAB text orientation)**UP****DOWN**

The **UP** and **DOWN** attributes of a SHEET control specify the orientation of the text on the TAB controls. UP (PROP:UP) specifies the TAB text appears vertical reading upwards, while DOWN (PROP:DOWN) specifies the TAB text appears vertical reading downwards. If both UP and DOWN attributes are present, the TAB text appears inverted (PROP:UpsideDown).

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
  SHEET,AT(0,0,320,175),USE(SelectedTab),RIGHT,DOWN !Tabs right reading down
  TAB('Tab One'),USE(?TabOne)
  PROMPT('Enter Data:'),AT(100,100,20,20),USE(?P1)
  ENTRY(@S8),AT(100,140,32,20),USE(E1)
  PROMPT('Enter More Data:'),AT(100,200,20,20),USE(?P2)
  ENTRY(@S8),AT(100,240,32,20),USE(E2)
  END
  PROMPT('Enter Data:'),AT(100,100,20,20),USE(?P3)
  ENTRY(@S8),AT(100,140,32,20),USE(E3)
  PROMPT('Enter More Data:'),AT(100,200,20,20),USE(?P4)
  ENTRY(@S8),AT(100,240,32,20),USE(E4)
  END
  BUTTON('Ok'),AT(100,180,20,20),USE(?Ok)
  BUTTON('Cancel'),AT(200,180,20,20),USE(?Cancel)
  END
```

USE (set field equate label or control update variable)

```
USE( | label | [,number] [,equate] )
      | variable |
```

USE	Specifies a variable or field equate label.
<i>label</i>	A field equate label to reference the control or structure in executable code. This must begin with a question mark (?) and meet all the requirements of a valid Clarion label.
<i>variable</i>	The label of the variable to receive the value the user enters in the control. The variable's label (with a leading question mark - <i>?VariableLabel</i>) becomes the field equate label for the control, unless the <i>equate</i> parameter is used.
<i>number</i>	An integer constant that specifies the number the compiler equates to the field equate label for the control (PROP:Feq, equivalent to {PROP:USE,2}).
<i>equate</i>	A field equate label to reference the control in executable code when the named <i>variable</i> has already been used in the same structure. This provides a mechanism to provide a unique field equate when the <i>variable</i> would not.

The **USE** attribute (PROP:USE) specifies a field equate label for the control or structure, or a variable for the control to update.

USE with a *label* parameter simply provides a mechanism for executable source code statements to reference the control or structure. USE with a *variable* parameter supplies the control with a variable to update by operator entry (on a window control) or to provide the value to print (on a report control).

The USE attribute's *number* parameter allows you to specify the actual field number the compiler assigns to the control. This *number* also is used as the new starting point for subsequent field numbering for controls without a *number* parameter in their USE attribute. Subsequent controls without a *number* parameter in their USE attribute are incremented (or decremented) relative to the last *number* assigned.

Two or more controls with exactly the same USE *variable* in one WINDOW or APPLICATION structure would create the same Field Equate Label for all, therefore, when the compiler encounters this condition, all Field Equate Labels for that USE variable are discarded. This makes it impossible to reference any of these controls in executable code, preventing confusion about which control you really want to reference. It also allows you to deliberately create this condition to display the contents of the variable in multiple controls with different display pictures. You may eliminate this situation by using *equate* parameters on these controls.

Writing to PROP:USE changes the USE attribute to use the name of the variable assigned. Reading it returns the contents of the current USE variable. PROP:Feq sets and returns the field number for the control.

Window Usage

Some controls or structures only allow a field equate *label* as the USE parameter, not a *variable*. These are: PROMPT, IMAGE, LINE, BOX, ELLIPSE, GROUP, RADIO, REGION, MENU, BUTTON, and TOOLBAR.

USE with a *variable* parameter supplies the control with a variable to update by operator entry. This is applicable to an ITEM with the CHECK attribute, ENTRY, OPTION, SPIN, TEXT, LIST, COMBO, CHECK, and CUSTOM.

PROP:ListFeq is equivalent to {PROP:USE,3} and sets the field equate label for the list portion of a COMBO control or a LIST control with the DROP attribute.

PROP:ButtonFeq is equivalent to {PROP:USE,4} and sets the field equate label for the drop button portion of a COMBO control or a LIST control with the DROP attribute.

Report Usage

Some controls and structures only allow a field equate *label* as the USE parameter, not a *variable*. These are: IMAGE, LINE, BOX, ELLIPSE, GROUP, RADIO, FORM, BREAK, DETAIL, HEADER, and FOOTER.

USE with a *variable* parameter supplies the control with a variable to update by operator entry. This is applicable to an OPTION, TEXT, LIST, CHECK, or CUSTOM. STRING controls may use either a field equate *label* or *variable*.

All controls and structures in a REPORT are automatically assigned numbers by the compiler. By default, these numbers start at one (1) and increment by one (1) for each control in the REPORT. The USE attribute's *number* parameter allows you to specify the actual field number the compiler assigns to the control or structure. This *number* also is used as the new starting point for subsequent numbering for controls and structures without a *number* parameter in their USE attribute. Subsequent controls and structures without a *number* parameter in their USE attribute are incremented relative to the last *number* assigned.

Example:

```

MainWin APPLICATION('My Application'),SYSTEM,MAX,ICON('MyIcon.ICO'),STATUS
  MENUBAR
    MENU('&File'),USE(?FileMenu)
    ITEM('&Open...'),USE(?OpenFile)
    ITEM('&Close'),USE(?CloseFile),DISABLE
    ITEM('E&xit'),USE(?MainExit)
  END
  END
  TOOLBAR,USE(?Toolbar)
  BUTTON('Exit'),USE(?MainExitButton)
  ENTRY(@S8),AT(100,160,20,20),USE(E2)
  ENTRY(@S8),AT(100,200,20,20),USE(E3,100)      !Field number 100
  ENTRY(@S8),AT(100,240,20,20),USE(E2,,?Number2:E2)  !
  END
  END

CustRpt REPORT,AT(1000,1000,6500,9000),THOUS
Detail  DETAIL,AT(0,0,6500,1000),USE(?Detail)      !Line item detail
STRING('Group Total:'),AT(5500,500,1500,500),USE(?Constant)!Field equate label
STRING(@N$11.2),AT(6000,1500,500,500),USE(Pre:F1) !USE variable
  END
  END

CODE
OPEN(MainWin)
DISABLE(?E2)      !Disable first entry control
DISABLE(100)     !Disable second entry control
DISABLE(?Number2:E2) !Disable third entry control
PrintRpt(CustRpt,?Detail) !Pass report and detail equate to print proc
  ACCEPT
  END

PrintRpt PROCEDURE(RptToPrint,DetailNumber)
CODE
OPEN(RptToPrint)      !Open passed report
PRINT(RptToPrint,DetailNumber) !Print its detail
CLOSE(RptToPrint)     !Close passed report

```

See Also:

Field Equate Labels

VALUE (set RADIO or CHECK control USE variable assignment)

```
VALUE( | string | )
      | truevalue , falsevalue |
```

VALUE Specifies the value assigned to the OPTION structure's USE variable when the RADIO control is selected by the user, or the values assigned to the CHECK control's USE variable when checked and un-checked by the user.

string A string constant containing the value to assign to the OPTION's USE variable.

truevalue A string constant containing the value to assign to the CHECK's USE variable when the user has checked the box (PROP:TrueValue, equivalent to {PROP:Value,1}).

falsevalue A string constant containing the value to assign to the CHECK's USE variable when the user has un-checked the box (PROP:FalseValue, equivalent to {PROP:Value,2}).

The **VALUE** attribute (PROP:VALUE) on a RADIO control specifies the value that is automatically assigned to the OPTION structure's USE variable when the RADIO control is selected by the user. This attribute overrides the RADIO control's *text* parameter.

The VALUE attribute on a CHECK control specifies the values that are automatically assigned to the control's USE variable when the user checks and un-checks the box. This attribute overrides the default assignment of zero and one.

All automatic type conversion rules apply to the values assigned to the control's USE variable. Therefore, if the *string*, *truevalue*, or *falsevalue* contains only numeric data and the USE variable is a numeric data type, it receives the numeric value.

PROP:VALUE may also be used on an ENTRY, SPIN, or COMBO control to interrogate the value that would be placed into the control's USE variable by UPDATE (or when the control loses focus) without actually updating the USE variable. This can cause EVENT:Rejected to generate, if appropriate.

Example:

```
Win WINDOW,AT(0,0,180,400)
  OPTION('Option 1'),USE(OptVar1),MSG('Pick One or Two')
  RADIO('Radio 1'),AT(120,0,20,20),USE(?R1),VALUE('10') !OptVar1 gets 10
  RADIO('Radio 2'),AT(140,0,20,20),USE(?R2),VALUE('20') !OptVar1 gets 20
  END
  OPTION('Option 2'),USE(OptVar2),MSG('Pick One or Two')
  RADIO('Radio 1'),AT(120,0,20,20),USE(?R1),VALUE('10') !OptVar2 gets '10'
  RADIO('Radio 2'),AT(140,0,20,20),USE(?R2),VALUE('20') !OptVar2 gets '20'
  END
  CHECK('Check 1'),AT(160,0),USE(Check1),VALUE('T','F')
END
```

VCR (set VCR control)**VCR**([*field*])

VCR Places Video Cassette Recorder (VCR) style buttons on a LIST or COMBO control.

field A field equate label that specifies the ENTRY control to use as a locator for a LIST or COMBO (PROP:VcrFeq, equivalent to {PROP:VCR,2}). This ENTRY control must appear before the LIST or COMBO control in the WINDOW structure.

The **VCR** attribute (PROP:VCR) places **V**ideo **C**assette **R**ecorder (VCR) style buttons on a LIST or COMBO control. The VCR style buttons affect the scrolling characteristics of the data displayed in the LIST or COMBO. There are seven buttons displayed as the VCR:

```

|<  Top of list      (EVENT:ScrollTop)
<<  Page Up         (EVENT:PageUp)
<   Entry Up        (EVENT:ScrollUp)
?   Locate          (EVENT:Locate)
>   Entry Down      (EVENT:ScrollDown)
>>  Page Down       (EVENT:PageDown)
>|  Bottom of list  (EVENT:ScrollBottom)

```

The *field* parameter names the control to get focus when the user presses the ? button. When the user enters data and then presses TAB on the locator *field*, the LIST or COMBO scrolls to its closest matching entry. If no *field* parameter is named, the ? button still appears but does nothing. To avoid even displaying the ? button, you may set PROP:VCR to TRUE instead of adding the VCR attribute to the LIST or COMBO control declaration.

Example:

```

MDIC WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
LIST,AT(140,0,20,20),USE(?L1),FROM(Q),HVSCROLL
ENTRY(@S8),AT(100,200,20,20),USE(E2)           !Locator control for L2
LIST,AT(140,100,20,20),USE(?L2),FROM(Q),HVSCROLL,VCR(?E2)!VCR with Locator active
END

CODE
OPEN(MDIC)
?L1{PROP:VCR} = TRUE           !VCR buttons without the ? button
ACCEPT
END

```

VERTICAL (set vertical progress bar display)**VERTICAL**

The VERTICAL (PROP:Vertical) attribute allows the progress control to operate from the bottom of the control to the top. If your progress control is positioned in a horizontal (left to right) display format, you should resize the progress control accordingly. Valid only for a PROGRESS control.

WALLPAPER (set background image)**WALLPAPER**(*image*)**WALLPAPER** Specifies a background image to display in the toolbar or window's client area.*image* A string constant specifying the name of the file to display.

The **WALLPAPER** attribute (PROP:WALLPAPER) specifies displaying the *image* as a background for the toolbar or window's client area. The image is stretched to fill the entire toolbar or window's client area unless either the **TILED** or **CENTERED** attributes are present.

Example:

```

MDIChild WINDOW('Child One'),MDI,SYSTEM,MAX
    MENUBAR
        MENU('Edit'),USE(?EditMenu)
        ITEM('Undo'),USE(?UndoText),KEY(CtrlZ),STD(STD:Undo)
        ITEM('Cu&t'),USE(?CutText),KEY(CtrlX),STD(STD:Cut)
        ITEM('Copy'),USE(?CopyText),KEY(CtrlC),STD(STD:Copy)
        ITEM('Paste'),USE(?PasteText),KEY(CtrlV),STD(STD:Paste)
    END
END
TOOLBAR,USE(?Toolbar),WALLPAPER('MyWall.GIF')
    BUTTON('Cut'),USE(?CutButton),ICON(ICON:Cut),STD(STD:Cut),FLAT
    BUTTON('Copy'),USE(?CopyButton),ICON(ICON:Copy),STD(STD:Copy),FLAT
    BUTTON('Paste'),USE(?PasteButton),ICON(ICON:Paste),STD(STD:Paste),FLAT
END
END

WinOne WINDOW,AT(,,380,200),MDI,WALLPAPER('MyWin.GIF')
    END

```

See Also:

CENTERED**TILED**

WITHNEXT (set widow elimination)

WITHNEXT([*siblings*])

WITHNEXT	Specifies the structure is always printed on the same page as print structures PRINTed immediately following it.
<i>siblings</i>	An integer constant or constant expression that specifies the number of following print structures to print on the same page. If omitted, the default value is one.

The **WITHNEXT** attribute (PROP:WITHNEXT) specifies that the DETAIL, or group HEADER or FOOTER structure (contained within a BREAK structure), is always printed on the same page as the specified number of print structures PRINTed immediately following it. This ensures that the structure is never printed on a page by itself, eliminating "widow" print structures. A "widow" print structure is defined as a group header, or first detail item in a related group of items, printed on the preceding page, separated from the rest of its related items.

The *siblings* parameter, if present, sets the number of following print structures that must be printed on the same page with the structure. To be counted, the following print structures must come from the same, or nested, BREAK structures. They must be related items. Any print structures not within the same, or nested, BREAK structures are printed but not counted as part of the required number of *siblings*.

Example:

```
CustRpt  REPORT
Break1   BREAK(SomeVariable)
         HEADER,WITHNEXT(2)    !Always print with 2 siblings
         !structure elements
         END
CustDetail DETAIL,WITHNEXT()   !Always print with 1 sibling
         !structure elements
         END
         FOOTER
         !structure elements
         END
         END
         END
```

WITHPRIOR (set orphan elimination)**WITHPRIOR**([*siblings*])

WITHPRIOR	Specifies the structure is always printed on the same page as print structures PRINTed immediately preceding it.
<i>siblings</i>	An integer constant or constant expression that specifies the number of preceding print structures to print on the same page. If omitted, the default value is one.

The **WITHPRIOR** attribute (PROP:WITHPRIOR) specifies that the DETAIL, or group HEADER or FOOTER structure (contained within a BREAK structure), is always printed on the same page as the specified number of print structures PRINTed immediately preceding it. This ensures that the structure is never printed on a page by itself, eliminating "orphan" print structures. An "orphan" print structure is defined as a group footer, or last detail item in a related group of items, that is printed on the following page separated from the rest of its related items.

The *siblings* parameter, if present, sets the number of preceding print structures that must be printed on the same page with the structure. To be counted, the preceding print structures must come from the same, or nested, BREAK structures. They must be related items. Any print structures not within the same, or nested, BREAK structures are printed, but not counted as part of the required number of *siblings*.

Example:

```

CustRpt  REPORT
Break1   BREAK(SomeVariable)
         HEADER
         !structure elements
         END
CustDetail  DETAIL,WITHPRIOR()      !Always print with 1 sibling
         !structure elements
         END
         FOOTER,WITHPRIOR(2)      !Always print with 2 siblings
         !structure elements
         END
         END
         END
         END

```

WIZARD (set "tabless" SHEET control)**WIZARD**

The **WIZARD** attribute (PROP:WIZARD) specifies a SHEET control that does not display its TAB controls. This allows the program to direct the user through each TAB in a specified sequence (usually with "Next" and "Previous" buttons).

ZOOM (set OLE object zooming)

The **ZOOM** attribute (PROP:ZOOM, write-only) specifies the OLE object stretches to fill the size specified by the OLE container control's AT attribute while maintaining the object's aspect ratio.

10 - Expressions

Overview

An expression is a mathematical, string, or logical formula that produces a value. An expression may be the source variable of an assignment statement, a parameter of a procedure, a subscript of an array (a dimensioned variable), or the condition of an IF, CASE, LOOP, or EXECUTE structure. Expressions may contain constant values, variables, and procedures which return values, all connected by logical and/or arithmetic or string operators.

Expression Evaluation

Expressions are evaluated in the standard algebraic order of operations. The precedence of operations is controlled by operator type and placement of parentheses. Each operation produces an (internal) intermediate value used in subsequent operations. Parentheses may be used to group operations within expressions. Expressions are evaluated beginning with the inner-most set of parentheses and working through to the outer-most set.

Precedence levels for expression evaluation, from highest to lowest, and left-to-right within each level, are:

Level 1 ()	Parenthetical Grouping
Level 2 -	Unary Minus (Negative sign)
Level 3 procedure call	Gets the RETURN value
Level 4 ^	Exponentiation
Level 5 * / %	Multiplication, Division, Modulus Division
Level 6 + -	Addition, Subtraction
Level 7 &	Concatenation
Level 8 = <>	Logical Comparisons
Level 9 NOT, AND, OR/XOR	Boolean expressions

Expressions may produce numeric values, string values, or logical values (true/false evaluation). An expression may contain no operators at all; it may be a single variable, constant value, or procedure call which returns a value.

Operators

Arithmetic Operators

An arithmetic operator combines two operands arithmetically to produce an intermediate value.

The operators are:

- +** Addition ($A + B$ gives the sum of A and B)
- Subtraction ($A - B$ gives the difference of A and B)
- *** Multiplication ($A * B$ multiples A by B)
- /** Division (A / B divides A by B)
- ^** Exponentiation ($A ^ B$ raises A to power of B)
- %** Modulus Division ($A \% B$ gives the remainder of A divided by B)

The Concatenation Operator

The ampersand (&) concatenation operator is used to append one string or string variable to another. The length of the resulting string is the sum of the lengths of the two values being concatenated. Numeric data types may be concatenated with strings or other numeric variables or constants. In many cases, the CLIP procedure should be used to remove any trailing spaces from a string being concatenated to another string.

Example:

```
CLIP(FirstName) & ' ' & Initial & '. ' & LastName    !Concatenate full name  
'SoftVelocity Corporation' & ', Inc.'            !Concatenate two constants
```

See Also:

CLIP

Numeric Expressions

Data Conversion Rules

FORMAT

Logical Operators

A logical operator compares two operands or expressions and produces a true or false condition. There are two types of logical operators: conditional and Boolean. Conditional operators compare two values or expressions. Boolean operators connect string, numeric, or logical expressions together to determine true-false logic. Operators may be combined to create complex operators.

Conditional Operators	=	Equal sign
	<	Less than
	>	Greater than
Boolean Operators	NOT	Boolean (logical) NOT
	~	Tilde (logical NOT)
	AND	Boolean AND
	OR	Boolean OR
	XOR	Boolean eXclusive OR
Combined operators	<>	Not equal
	~=	Not equal
	NOT =	Not equal
	<=	Less than or equal to
	=<	Less than or equal to
	~>	Not greater than
	NOT >	Not greater than
	>=	Greater than or equal to
	=>	Greater than or equal to
	~<	Not less than
NOT <	Not less than	

During logical evaluation, any non-zero numeric value or non-blank string value indicates a true condition, and a null (blank) string or zero numeric value indicates a false condition.

Example:

<u>Logical Expression</u>	<u>Result</u>
A = B	True when A is equal to B
A < B	True when A is less than B
A > B	True when A is greater than B
A <> B, A ~= B, A NOT = B	True when A is not equal to B
A ~< B, A >= B, A NOT < B	True when A is not less than B
A ~> B, A <= B, A NOT > B	True when A is not greater than B
~ A, NOT A	True when A is null or zero
A AND B	True when A is true and B is true
A OR B	True when A is true, or B is true, or both are true
A XOR B	True when A is true or B is true, but not both.

Constants

Numeric Constants

Numeric constants are fixed numeric values. They may occur in data declarations, in expressions, and as parameters of procedures or attributes. A numeric constant may be represented in decimal (base 10--the default), binary (base 2), octal (base 8), hexadecimal (base 16), or scientific notation formats. Formatting characters, such as dollar signs and commas, are not permitted in numeric constants; only leading plus or minus signs and the decimal point are allowed.

Decimal (base ten) numeric constants may contain an optional leading minus sign (hyphen character), an integer, and an optional decimal with a fractional component. Binary (base two) numeric constants may contain an optional leading minus sign, the digits 0 and 1, and a terminating B or b character. Octal (base eight) numeric constants contain an optional leading minus sign, the digits 0 through 7, and a terminating O or o character. Hexadecimal (base sixteen) numeric constants contain an optional leading minus sign, the digits 0 through 9, alphabet characters A through F (representing the numbers 10 through 15) and a terminating H or h character. If the left-most character is a letter A through F, a leading zero must be used.

Example:

```
-924    !Decimal constants  
76.346  
+76.346
```

```
1011b  !Binary constants  
-1000110B
```

```
3403o  !Octal constants  
-7041312O
```

```
-1FFBh !Hexadecimal constants  
0CD1F74FH
```

String Constants

A string constant is a set of characters enclosed in single quotes (apostrophes). The maximum length of a string constant is 255 characters. Characters that cannot be entered from the keyboard may be inserted into a string constant by enclosing their ASCII character codes in angle brackets (<>). ASCII character codes may be represented in decimal, hexadecimal, binary, or octal numeric constant format.

In a string constant, a left angle bracket (<) initiates a scan for a right angle bracket. Therefore, to include a left angle bracket in a string constant requires two left angle brackets in succession. To include an apostrophe as part of the value inside a string constant requires two apostrophes in succession. Two apostrophes ("), with no characters (or just spaces) between them, represents a null, or blank, string. Consecutive occurrences of the same character within a string constant may be represented by *repeat count* notation. The number of times the character is to be repeated is placed within curly braces ({ }) immediately following the character to repeat. To include a left curly brace ({) as part of the value inside a string constant requires two left curly braces ({ {) in succession.

The ampersand (&) is always valid in a string constant. However, depending on the assignment's destination, it may be interpreted as an underscore for a hot letter (for example, a PROMPT control's display *text*). In this case, you double it up (&&) to end up with a single ampersand in the screen display.

Example:

```
'string constant'    !A string constant
'It''s a girl!'     !With embedded apostrophe
'<27,15>'           !Using decimal ASCII codes
'A << B'            !With embedded left angle, A < B
'{*20}'             !Twenty asterisks, repeat-count notation
''                  !A null (blank) string
```

Types of Expressions

Numeric Expressions

Numeric expressions may be used as parameters of procedures, the condition of IF, CASE, LOOP, or EXECUTE structures, or as the source portion of an assignment statement where the destination is a numeric variable. A numeric expression may contain arithmetic operators and the concatenation operator, but they may not contain logical operators. When used in a numeric expression, string constants and variables are converted to numeric intermediate values. If the concatenation operator is used, the intermediate value is converted to numeric after the concatenation occurs.

Example:

```
Count + 1           !Add 1 to Count
(1 - N * N) / R     !N times N subtracted from 1 then divided by R
305 & 7854555      !Concatenate area code with phone number
```

See Also:

Data Conversion Rules

String Expressions

String expressions may be used as parameters of procedures and attributes, or as the source portion of an assignment statement when the destination is a string variable. String expressions may contain a single string or numeric variable, or a complex combination of sub-expressions, procedures, and operations.

Example:

```
StringVar   STRING(30)
Name       STRING(10)
Weight     STRING(3)
Phone      LONG
          CODE
!Concatenate a constant and variable
StringVar = 'Address:' & Cus:Address

!Concatenate constant values
!and FORMAT procedure's return value
StringVar = 'Phone:' & ' 305-' & FORMAT(Phone,@P###-####P)

!Concatenate a constant and variable
StringVar = Weight & 'lbs.'
```

See Also:

CLIP

The Concatenation Operator

Data Conversion Rules

FORMAT

Logical Expressions

Logical expressions evaluate true-false conditions in IF, LOOP UNTIL, and LOOP WHILE control structures. Control is determined by the final result (true or false) of the expression. Logical expressions are evaluated from left to right. The right operand of an AND, OR, or XOR logical expression will only be evaluated if it could affect the result. Parentheses should be used to eliminate ambiguous evaluation and to control evaluation precedence. The level or precedence for the logical operators is as follows:

Level 1 Conditional operators
Level 2 ~, NOT
Level 3 AND
Level 4 OR, XOR

Example:

```
LOOP UNTIL KEYBOARD()      !True when user presses any key
  !some statements
END
```

```
IF A = B THEN RETURN.      !RETURN if A is equal to B
```

```
LOOP WHILE ~ Done#        !Loop while false (Done# = 0)
  !some statements
END
```

```
IF A >= B OR (C > B AND E = D) THEN RETURN.
!True if a >= b, also true if
!both c > b and e = d.
!The second part of the expression
!(after OR) is evaluated only if the
!first part is not true.
```

See Also:

IF (conditional execution structure)

LOOP

Property Expressions

[*target*] [\$] [*control*] { *property* [, *element*] }

<i>target</i>	The label of an APPLICATION, WINDOW, REPORT, VIEW, or FILE structure, the label of a BLOB, or one of the built-in variables: TARGET, PRINTER, or SYSTEM. If omitted, TARGET is assumed.
\$	Required delimiter when both <i>target</i> and <i>control</i> are specified. Omit if either <i>target</i> or <i>control</i> is omitted.
<i>control</i>	A field number or field equate label for the control in the <i>target</i> structure (APPLICATION, WINDOW, or REPORT) to affect. If omitted, the <i>target</i> must be specified. The <i>control</i> must be omitted if the <i>target</i> is a FILE, BLOB, or the PRINTER or SYSTEM built-in variables.
<i>property</i>	An integer constant, EQUATE, or variable that specifies the property (attribute) to change. It can also be a string when referencing an OCX or OLE container property.
<i>element</i>	An integer constant or variable that specifies which element to change (for <i>properties</i> which are arrays).

This property expression syntax allows you access to all the attributes (properties) of APPLICATION, WINDOW, or REPORT structures, or any control within these structures. To specify an attribute of an APPLICATION, WINDOW, REPORT, VIEW, or FILE structure (not a component control), omit the *control* portion of the property expression. To specify a control in the current window, omit the *target* portion of the property expression.

REPORT data structures are never the *target* by default. Therefore, either SETTARGET must be used to change the *target* to the REPORT, or the REPORT structure's label must be explicitly specified as the *target* before you can change any property of the structure, or any control it contains.

Property expressions may be used in Clarion language statements anywhere a string expression is allowed, or as the destination or source of simple assignment statements. They may not be used in operating assignment statements (such as +=, *=, etc.). Assigning a new value to a property is a simple assignment with the property as the destination and the new value as the source. Determining the current value of a property is a simple assignment where the property is the source and the variable to receive its value is the destination. A Property expression may also be used as an executable statement (without an assignment statement) when the property expression is a method call for an OLE or OCX control.

All properties are treated as string data at runtime; the compiler automatically performs any necessary data type conversion. Any property without parameters is binary (toggle). Binary properties are either "present" or "missing" and return a '1' if present, and " (null) if missing. Changing the value of a binary property to " (null), '0' (zero), or any non-numeric string sets it to missing. Changing it to any other value sets it to "present."

Most properties can be both examined (read) and changed (written). However, some properties are "read-only" and cannot be changed. Assigning a value to a "read-only" property has no effect at all. Other properties are "write-only" properties that are meaningless if read. Some properties are arrays that contain multiple values. The syntax for addressing a particular property array *element* uses a comma (not square brackets) as the delimiter between the *property* and the *element* number.

Built-in Variables

There are three built-in variables in the Clarion for Windows runtime library: TARGET, PRINTER, and SYSTEM. These are only used with the property assignment syntax to identify the *target* in a property expression.

TARGET normally references the window that currently has focus. It can also be set to reference a window in another execution thread or the currently printing REPORT, enabling you to affect the properties of controls and windows in other execution threads and dynamically change report control properties while printing. The SETTARGET statement and SYSTEM {PROP:Target} property changes the TARGET variable's reference.

PRINTER references the Printer Properties (only) to be used by the next REPORT opened (and any subsequent reports).

SYSTEM specifies global properties used by the entire application. There are a number of runtime properties that may use the SYSTEM variable to set or query application-wide properties.

Example:

```
MainWin APPLICATION('My Application'),SYSTEM,MAX,ICON('MyIcon.ICO'),STATUS,RESIZE
  MENUBAR
    MENU('File'),USE(?FileMenu)
      ITEM('Open...'),USE(?OpenFile)
      ITEM('Close'),USE(?CloseFile),DISABLE
      ITEM('E&xit'),USE(?MainExit)
    END
    MENU('Help'),USE(?HelpMenu)
      ITEM('Contents'),USE(?HelpContents),STD(STD:HelpIndex)
      ITEM('Search for Help On...'),USE(?HelpSearch),STD(STD:HelpSearch)
      ITEM('How to Use Help'),USE(?HelpOnHelp),STD(STD:HelpOnHelp)
      ITEM('About MyApp...'),USE(?HelpAbout)
    END
  END
  TOOLBAR
    BUTTON('Open'),USE(?OpenButton),ICON(ICON:Open)
  END
END
```

```

CODE
OPEN(MainWin)
MainWin{PROP:Text} = 'A New Title'           !Change window title
?OpenButton{PROP:ICON} = ICON:Asterisk     !Change button icon
?OpenButton{PROP:AT,1} = 5                 !Change button x position
?OpenButton{PROP:AT,2} = 5                 !Change button y position
IF MainWin$?HelpContents{PROP:STD} <> STD:HelpIndex
  MainWin$?HelpContents{PROP:STD} = STD:HelpIndex
END
MainWin{PROP:MAXIMIZE} = 1                 !Expand to full screen

ACCEPT
CASE ACCEPTED()                           !Which control was chosen?
OF ?OpenFile                               !Open... menu selection
OROF ?OpenButton                           !Open button on toolbar
  START(OpenFileProc)                     !Start new execution thread
OF ?MainExit                               !Exit menu selection
OROF ?MainExitButton                       !Exit button on toolbar
  BREAK                                   !Break ACCEPT loop
OF ?HelpAbout                              !About... menu selection
  HelpAboutProc                           !Call application information procedure
END
END
CLOSE(MainWin)                             !Close APPLICATION
RETURN

```

See Also:

SETTARGET

Runtime Properties

Runtime Expression Evaluation

Clarion has the ability to evaluate expressions dynamically created at runtime, rather than at development time. This allows a Clarion program to construct expressions "on the fly." This also makes it possible to allow an end-user to enter the expression to evaluate.

An expression is a mathematical or logical formula that produces a value; it is not a complete Clarion language statement. Expressions may only contain constant values, variables, or procedure calls which return a value, all connected by logical and/or arithmetic operators. An expression may be used as the source side of an assignment statement, a parameter of a procedure, a subscript of an array (a dimensioned variable), or the conditions of IF, CASE, LOOP, or EXECUTE structures.

Any program variable, and most of the internal Clarion procedures, can be used as part of a runtime expression string. User-defined procedures that fall within certain specific guidelines (described in the BIND statement documentation) may also be used in runtime expression strings.

All of the standard Clarion expression syntax is available for use in runtime expression strings. This includes parenthetical grouping and all the arithmetic, logical, and string operators. Dynamic expressions are evaluated just as any other Clarion expression and all the standard operator precedence level rules described in the Expression Evaluation section (see page 3) apply.

It takes three steps to use runtime expression strings:

- The variables that are allowed to be used in the expressions must be explicitly declared with the BIND statement.
- The expression must be built. This may involve concatenating user choices or allowing the user to directly type in their own expression.
- The expression is passed to the EVALUATE procedure which returns the result. If the expression is not a valid Clarion expression, ERRORCODE is set.

Once the expression is evaluated, its result is used just as the result of any hard-coded expression would be. For example, a runtime expression string could provide a filter expression to eliminate certain records when viewing or printing a database (the FILTER expression of a VIEW structure is an implicit runtime expression string).

See Also:

BIND, EVALUATE, POPBIND, PUSHBIND, UNBIND

BIND (declare runtime expression string variable)

```

BIND( | name,variable | )
        | name,procedure |
        | group          |

```

BIND	Identifies variables allowed to be used in dynamic expressions.
<i>name</i>	A string constant containing the identifier used in the dynamic expression. This may be the same as the <i>variable</i> or <i>procedure</i> label.
<i>variable</i>	The label of any variable (including fields in FILE, GROUP, or QUEUE structures) or passed parameter. If it is an array, it must have only one dimension.
<i>procedure</i>	The label of a Clarion language PROCEDURE which returns a STRING, REAL, or LONG value. If parameters are passed to the <i>procedure</i> , they must be STRING value-parameters (passed by value, not by address) and may not be omissible.
<i>group</i>	The label of a GROUP, RECORD, or QUEUE structure declared with the BINDABLE attribute.

The **BIND** statement declares the logical name used to identify a variable, EQUATE, or user-defined procedure in runtime expression strings. A variable or user-defined procedure must be identified with the BIND statement before it can be used in an expression string for either the EVALUATE procedure or a VIEW structure's FILTER attribute.

BIND(*name,variable*)

The specified *name* is used in the expression in place of the label of the *variable*.

BIND(*name,procedure*)

The specified *name* is used in the expression in place of the label of the *procedure*.

BIND(*group*)

Declares all the variables within the GROUP, RECORD, or QUEUE (with the BINDABLE attribute) available for use in a dynamic expression. The contents of each variable's NAME attribute is the logical name used in the dynamic expression. If no NAME attribute is present, the label of the variable (including prefix) is used.

A GROUP, RECORD, or QUEUE structure declared with the BINDABLE attribute has space allocated in the .EXE for the names of all of the data elements in the structure. This creates a larger program that uses more memory than it normally would. Also, the more variables that are bound at one time, the slower the EVALUATE procedure will work. Therefore, **BIND**(*group*) should only be used when a large proportion of the constituent fields are going to be used.

Example:

```

PROGRAM
  MAP
    AllCapsFunc (STRING),STRING           !Clarion procedure
  END
Header    FILE,DRIVER('Clarion'),PRE(Hea),BINDABLE !Declare header file layout
OrderKey  KEY(Hea:OrderNumber)
Record    RECORD
OrderNumber LONG
ShipToName STRING(20)
          END
          END

```

```

StringVar STRING(20)
CODE
  BIND('ShipName',Hea:ShipToName)
  BIND('SomeFunc',AllCapsFunc)
  BIND('StringVar',StringVar)
  StringVar = 'SMITH'
  CASE EVALUATE('StringVar = SomeFunc(ShipName)')
  OF ''
    IF ERRORCODE()
      MESSAGE('Error ' & ERRORCODE() & ' -- ' & ERROR())
    ELSE
      MESSAGE('Unkown error evaluating expression')
    END
  OF '0'
    DO NonSmithProcess
  OF '1'
    DO SmithProcess
  END

```

```

AllCapsFunc PROCEDURE(PassedString)
CODE
  RETURN(UPPER(PassedString))

```

See Also:

UNBIND

BINDEXPRESSION

EVALUATE

PUSHBIND

POPBIND

FILTER

BINDEXPRESSION(declare runtime expression)

BINDEXPRESSION(| *name*,*expression* |)

BINDEXPRESSION

Identifies variables allowed to be used in dynamic expressions.

name A string constant containing the identifier used in the dynamic expression.

expression A mathematical formula containing any valid combination of variables, functions, operators, and constants.

The **BINDEXPRESSION** statement declares the logical name used to identify an expression used in runtime expression strings.

The **EVALUATE** statement must parse an expression every time it is called. If some expression is evaluated frequently, this can be time consuming.

BINDEXPRESSION parses the expression only once and is saved in a pre-compiled form under the given *name*. Later, this expression can be used in EVALUATE as a stand alone, or as a part of another expression in EVALUATE, or in another BINDEXPRESSION.

Example:

```

BIND('Var', Var)
BINDEXPRESSION('Power2', 'Var * Var')
...
Var = 10
P2 = EVALUATE ('Power2')
...
BINDEXPRESSION ('Circle', '3.14159 * Power2')
BINDEXPRESSION('color:red',color:red)
MESSAGE('color:red = ' & color:red & '|Eval(color:red)= ' &|
evaluate('color:red') )                               !Shows same number twice

BINDEXPRESSION('Match:Soundex',Match:Soundex)
FILTER('MATCH(Cus:Name,NameWanted,Match:Soundex)')
```

See Also:

BIND (declare runtime expression string variable)

EVALUATE (return runtime expression string result)

EVALUATE (return runtime expression string result)

EVALUATE(*expression*)

EVALUATE Evaluates runtime expression strings.

expression A string constant or variable containing the expression to evaluate.

The **EVALUATE** procedure evaluates the *expression* and returns the result as a STRING value. If the *expression* does not meet the rules of a valid Clarion expression, the result is a null string (""), and ERRORCODE is set. A logical *expression* returns a string containing either zero ('0') or one ('1'), while an arithmetic *expression* returns the actual result of the *expression* (in a string). To use less than (<) in the *expression*, you must double it up (<<) to prevent compiler errors. The more variables are bound at one time, the slower the EVALUATE procedure works. Therefore, BIND(*group*) should only be used when most of the *group's* fields are needed, and UNBIND should be used to free all variables and user-defined procedures not currently required for use in dynamic expressions. **PATH()**, **SHORTPATH()** and **LONGPATH()** can now be used in expressions evaluated by the EVALUATE statements in runtime application and in the templates.

Field Qualification syntax (dot syntax) cannot be used in the *expression*. Variables should be binded using the standard prefix notation.

Return Data Type: **STRING**

Errors Posted: 1010 – Illegal Expression
1011 – Variable Not Found
1012 – Mismatched POPBIND

Example:

```

MAP
AllCapsFunc PROCEDURE(STRING),STRING           !Clarion procedure
END
Header      FILE,DRIVER('Clarion'),PRE(Hea),BINDABLE !Declare header file layout
OrderKey    KEY(Hea:OrderNumber)
Record      RECORD
OrderNumber LONG
ShipToName  STRING(20)
            END
            END

```

```

StringVar   STRING(20)
CODE
BIND('ShipName',Hea:ShipToName)
BIND('SomeFunc',AllCapsFunc)
BIND('StringVar',StringVar)
StringVar = 'SMITH'
CASE EVALUATE('StringVar = SomeFunc(ShipName)')
OF ''
  IF ERRORCODE()
    MESSAGE('Error ' & ERRORCODE() & ' -- ' & ERROR())
  END
OF '0'
  DO NonSmithProcess
OF '1'
  DO SmithProcess
END

```

```

AllCapsFunc PROCEDURE(PassedString)
CODE
RETURN(UPPER(PassedString))

```

See Also:

BIND

UNBIND

PUSHBIND

POPBIND

FILTER

POPBIND (restore runtime expression string name space)

POPBIND

The **POPBIND** statement restores the previous BIND statement's *name* space for variables and procedures previously bound. This restores the previous scope used by previous BIND statements.

Example:

```
SomeProc PROCEDURE
```

```
OrderNumber  LONG
Item          LONG
Quantity     SHORT
```

```
CODE
```

```
BIND('OrderNumber',OrderNumber)
```

```
BIND('Item',Item)
```

```
BIND('Quantity',Quantity)
```

```
AnotherProc                                !Call another procedure
```

```
UNBIND('OrderNumber',OrderNumber)
```

```
UNBIND('Item',Item)
```

```
UNBIND('Quantity',Quantity)
```

```
AnotherProc PROCEDURE
```

```
OrderNumber  LONG
Item          LONG
Quantity     SHORT
```

```
CODE
```

```
PUSHBIND                                !Create new scope for BIND
```

```
BIND('OrderNumber',OrderNumber) !Bind variables with same names in new scope
```

```
BIND('Item',Item)
```

```
BIND('Quantity',Quantity)
```

```
!Do some Processing
```

```
UNBIND('OrderNumber')
```

```
UNBIND('Item')
```

```
UNBIND('Quantity')
```

```
POPBIND                                !Restore previous scope for BIND
```

See Also: PUSHBIND, EVALUATE

PUSHBIND (save runtime expression string name space)**PUSHBIND**([*clearflag*])**PUSHBIND** Creates a new scope for subsequent BIND statements.

clearflag An integer constant or variable containing either zero (0) or one (1). When zero, the BIND statement's *name* space is cleared of all variables and procedures previously bound. When one, all variables and procedures previously bound are left in place. If omitted, the *clearflag* is zero.

The **PUSHBIND** statement creates a new scope for subsequent BIND statements. This scope terminates with the next POPBIND statement. This creates a new scope for subsequent BIND statements, allowing you to create new BIND *names* for variables with the same *name* without creating conflicts with the *names* from a previous scope.

Example:

```

SomeProc PROCEDURE
OrderNumber LONG
Item         LONG
Quantity     SHORT
CODE
  BIND('OrderNumber',OrderNumber)
  BIND('Item',Item)
  BIND('Quantity',Quantity)
  AnotherProc                !Call another procedure
  UNBIND('OrderNumber',OrderNumber)
  UNBIND('Item',Item)
  UNBIND('Quantity',Quantity)

AnotherProc PROCEDURE
OrderNumber LONG
Item         LONG
Quantity     SHORT
CODE
  PUSHBIND                !Create new scope for BIND
  BIND('OrderNumber',OrderNumber) !Bind variables with same names in new scope
  BIND('Item',Item)
  BIND('Quantity',Quantity)

  !Do some Processing
  UNBIND('OrderNumber')
  UNBIND('Item')
  UNBIND('Quantity')
  POPBIND                !Restore previous scope for BIND

```

See Also: POPBIND, EVALUATE

UNBIND (free runtime expression string variable)UNBIND([*name*])**UNBIND** Frees variables from use in runtime expression strings.*name* A string constant that specifies the identifier used by the dynamic expression evaluator. If omitted, all bound variables are unbound.

The **UNBIND** statement frees logical names previously bound by the BIND statement. The more variables that are bound at one time, the slower the EVALUATE procedure works. Therefore, UNBIND should be used to free all variables and user-defined procedures not currently available for use in runtime expression strings.

Example:

```

PROGRAM
MAP
  AllCapsFunc(STRING),STRING           !Clarion procedure
END

Header      FILE,DRIVER('Clarion'),PRE(Hea) !Declare header file layout
AcctKey     KEY(Hea:AcctNumber)
OrderKey    KEY(Hea:OrderNumber)
Record      RECORD
AcctNumber  LONG
OrderNumber LONG
ShipToName  STRING(20)
ShipToAddr  STRING(20)
ShipToCity  STRING(20)
ShipToState STRING(20)
ShipToZip   STRING(20)
            END
            END

Detail      FILE,DRIVER('Clarion'),PRE(Dtl),BINDABLE !Bindable RECORD structure
OrderKey    KEY(Dtl:OrderNumber)
Record      RECORD
OrderNumber LONG
Item        LONG
Quantity    SHORT
            END
            END

```

```
CODE
BIND('ShipName',Hea:ShipToName)
BIND(Dt1:Record)
BIND('SomeFunc',AllCapsFunc)
UNBIND('ShipName')           !UNBIND the variable
UNBIND('SomeFunc')           !UNBIND the Clarion language procedure
UNBIND                        !UNBIND all bound variables
```

```
AllCapsFunc PROCEDURE(PassedString)
CODE
RETURN(UPPER(PassedString))
```

See Also:

BIND

EVALUATE

PUSHBIND

POPBIND

11 - Assignments

Simple Assignment Statements

destination = *source*

destination The label of a variable or runtime property.

source A numeric or string constant, variable, procedure, expression, or data structure property.

The = sign assigns the value of *source* to the *destination*; it copies the value of the *source* expression into the *destination* variable. If *destination* and *source* are different data types, the value the *destination* receives from the *source* is dependent upon the Data Conversion Rules.

Example:

```
StringVar  STRING(10)
LongVar    LONG
RealVar    REAL
CODE
StringVar = 'JONES'           !Variable = string constant
RealVar   = 3.14159           !Variable = numeric constant
RealVar   = SQRT(1 - Sine * Sine) !Variable = procedure return value
LongVar   = B + C + 3         !Variable = numeric expression
StringVar = CLIP(FirstName) & ' ' Initial & '. ' & LastName
                                           !Variable = string expression

StringVar = '10'              !Assign numeric data to string then
Longvar   = StringVar         !Automatic data conversion results
                                           !and Longvar will contain: 10
```

See Also:

Data Conversion Rules

Property Expressions

Operating Assignments

destination += *source*

destination -= *source*

destination *= *source*

destination /= *source*

destination ^= *source*

destination %= *source*

destination Must be the label of a variable. This may not be a runtime property.

source A constant, variable, procedure, or expression.

Operating assignment statements perform their operation on the *destination* and *source*, then assign the result to the *destination*. Operating assignment statements are more efficient than their functionally equivalent operations.

Example:

<u>Operating Assignment</u>	<u>Functional Equivalent</u>
A += 1	A = A + 1
A -= B	A = A - B
A *= -5	A = A * -5
A /= 100	A = A / 100
A ^= I + 1	A = A ^ (I + 1)
A %= 7	A = A % 7

See Also:

Data Conversion Rules

Property Expressions

Deep Assignment

destination ::= *source*

destination The label of a GROUP, RECORD, or QUEUE data structure, or an array.

source The label of a GROUP, RECORD, or QUEUE data structure, or a numeric or string constant, variable, procedure, or expression.

The ::= sign executes a deep assignment statement which performs multiple individual component variable assignments from one data structure to another. The assignments are only performed between the variables within each structure that have exactly matching labels, ignoring all prefixes. The compiler looks within nested GROUP structures to find matching labels. Any variable in the *destination* which does not have a label exactly matching a variable in the *source*, is not changed.

Deep assignments are performed just as if each matching variable were individually assigned to its matching variable. This means that all normal data conversion rules apply to each matching variable assignment. For example, the label of a nested *source* GROUP may match a nested *destination* GROUP or simple variable. In this case, the nested *source* GROUP is assigned to the *destination* as a STRING, just as normal GROUP assignment is handled.

The name of a *source* array may match a *destination* array. In this case, each element of the *source* array is assigned to its corresponding element in the *destination* array. If the *source* array has more or fewer elements than the *destination* array, only the matching elements are assigned to the *destination*.

If the *destination* is an array variable that is not part of a GROUP, RECORD, or QUEUE, and the *source* is a constant, variable, or expression, then each element of the *destination* array is initialized to the value of the *source*. This is a much more efficient method of initializing an array to a specific value than using a LOOP structure and assigning each element in turn.

The *destination* or *source* may also name a CLASS structure, which, in this case, will be treated as a GROUP. However, if you do so, you violate the concept of encapsulation, since deep assignment is a structure piercing operation. Therefore, it is not recommended.

Example:

```
Group1 GROUP
S      SHORT
L      LONG
      END
```

```
Group2 GROUP
L      SHORT
S      REAL
T      LONG
      END
```

```
ArrayField SHORT,DIM(1000)
```

```
CODE
```

```
Group2 ::= Group1      !Is equivalent to:
                        !Group2.S = Group1.S
                        !Group2.L = Group1.L
                        !and performs all necessary data conversion
```

```
ArrayField ::= 7      !Is equivalent to:
                      !LOOP I# = 1 to 1000
                      ! ArrayField[I#] = 7
                      !END
```

See Also:

GROUP

RECORD

QUEUE

DIM

Reference Assignments

destination &= *source*

<i>destination</i>	The label of a reference variable.
<i>source</i>	This may be: <ul style="list-style-type: none"> • The label of a variable or data structure of the same type as referenced by the <i>destination</i>. • The label of another reference variable of the same type as the <i>destination</i>. • A PROCEDURE which returns the data type the <i>destination</i> will receive. • An expression (yielding a LONG value, such as the return value of the ADDRESS procedure) that defines the memory address of a variable of the same type as referenced by the <i>destination</i> (which must be a reference to any simple data type except STRING, CSTRING, or PSTRING). • The NULL built-in variable.

The &= sign executes a reference assignment statement. A reference assignment statement assigns a reference to the *source* variable to the *destination* reference variable. When used in a conditional expression (such as an IF statement), a reference assignment statement determines reference equality (are the two reference variables "pointing at" the same thing?).

Depending upon the data type being referenced, the *destination* reference variable may receive the *source*'s memory address, or a more complex internal data structure (describing the location and type of *source* data).

When the *source* is the built-in variable NULL, the reference assignment statement may either clear the *destination* reference variable, or detect an unreferenced reference variable (when the reference assignment statement is placed in a conditional expression).

The declarations of the *destination* reference variable and its *source* must match exactly (unless the *destination* is declared as an ANY variable); reference assignment does not perform automatic type conversion. For example, a reference assignment statement to a *destination* declared as &QUEUE must have a *source* that is either another &QUEUE reference variable, the label of a QUEUE structure, or the ADDRESS(MyQueue) procedure. However, if the *destination* is a reference to a string (&STRING), the *source* may also be a data structure that is normally treated as string data when addressed as a single unit (GROUP, RECORD, QUEUE, MEMO).

Example:

```

Queue1  QUEUE
ShortVar  SHORT
LongVar1  LONG
LongVar2  LONG
        END

QueueRef  &QUEUE           !Reference a QUEUE, only
Queue1Ref &Queue1         !Reference to a QUEUE defined exactly as Queue1, only

LongRef   &LONG            !Reference a LONG, only

LongRef2  &LONG            !Reference a LONG, only

CODE
QueueRef  &= Queue1       !Assign QUEUE reference
Queue1Ref &= Queue1       !Assign QUEUE reference

IF Queue1Ref &= QueueRef  !Are they referencing the same QUEUE?
  MESSAGE('Both Pointing at same QUEUE')
END

IF SomeCondition          !Evaluate some condition
  LongRef &= Queue1.LongVar1 !and reference an appropriate variable
ELSE
  LongRef &= Queue1.LongVar2
END
LongRef += 1              !Increment either LongVar1 or LongVar2
                           !depending upon which variable is referenced

IF LongRef2 &= NULL       !Detect unreferenced reference variable and
  LongRef2 &= LongRef     !create a second reference to the same data
END

LongRef &= ADDRESS(Queue1.LongVar1) !Reference assign the address of
                                     !a simple data type

```

See Also:

Reference Variables

ANY

NEW

CLEAR (clear a variable)**CLEAR**(*label* [,*n*])

CLEAR	Clears the value from a variable.
<i>label</i>	The label of a variable (except BLOB types), GROUP, RECORD, QUEUE, CLASS, or FILE structure. If the variable has a DIM attribute, the entire array is cleared. A single element of an array cannot be CLEARed.
<i>n</i>	A numeric constant; either 1 or -1. If omitted, numeric variables are cleared to zero, STRING variables are cleared to spaces, and PSTRING and CSTRING variables are set to zero length.

The **CLEAR** statement clears the value from the *label* variable.

The presence of the *n* parameter indicates a cleared value other than zero or blank. If *n* is 1, the *label* variable is set to the highest possible value for that data type. For the STRING, PSTRING and CSTRING data types, that is all ASCII 255. If *n* is -1, the *label* variable is set to the lowest possible value for that data type. For the STRING data type, that is all ASCII zeroes (0). For the PSTRING and CSTRING data types, that is a zero length string.

If the *label* parameter names a GROUP, RECORD, or QUEUE structure, all variables in the structure are cleared and all reference variables in the structure are set to NULL. If the *label* parameter names a FILE structure and the *n* parameter is omitted, all variables in the FILE structure (including any MEMO and/or BLOB fields) are cleared. If the *label* parameter names a CLASS structure or an object derived from a CLASS, all variables in the object are cleared and all reference variables are set to NULL.

Example:

```

MyQue QUEUE
F1     LONG
F2     STRING(20)
F3     &CSTRING    !Reference to a CSTRING
F4     ANY         !ANY can be a reference variable to any simple data type
      END

CODE
CLEAR(MyQue)      !Equivalent to:
                  ! MyQue.F1 = 0
                  ! MyQue.F2 = ''
                  ! MyQue.F3 &= NULL
                  ! MyQue.F4 &= NULL

CLEAR(Count)     !Clear a variable
CLEAR(Cus:Record)!Clear the record structure
CLEAR(Customer)  !Clear the record structure and any memos and blobs
CLEAR(Amount,1)  !Clear variable to highest possible value
CLEAR(Amount,-1) !Clear variable to lowest possible value

```

See Also: Reference Assignment Statements, GROUP, RECORD, QUEUE, DIM, CLASS, ANY

Data Type Conversion Rules

The Clarion language provides automatic conversion between data types. However, some assignments can produce an unequal source and destination. Assigning an "out of range" value can produce unpredictable results.

See Also:

Base Types

BCD Operations and Procedures

Type Conversion and Intermediate Results

Simple Assignment Data Type Conversion

Base Types

To facilitate this automatic data type conversion, Clarion internally uses four Base Types to which all data items are automatically converted when any operation is performed on the data. These types are: STRING, LONG, DECIMAL, and REAL. These are all standard Clarion data types.

The STRING Base Type is used as the intermediate type for all string operations. The LONG, DECIMAL, and REAL Base Types are used in all arithmetic operations. Which numeric type is used, and when, is determined by the original data types of the operands and the type of operation being performed on them.

The "normal" Base Type for each data type is:

Base Type LONG:

BYTE
SHORT
USHORT
LONG
DATE
TIME
Integer Constants
Strings declared with @P pictures

Base Type DECIMAL:

ULONG
DECIMAL
PDECIMAL
STRING(@Nx.y)
Decimal Constants

Base Type REAL:

SREAL
REAL
BFLOAT4
BFLOAT8
STRING(@Ex.y)
Scientific Notation Constants
Untyped (? and *) Parameters

Base Type STRING:

STRING
CSTRING
PSTRING
String Constants

DATE and TIME data types are first converted to Clarion Standard Date and Clarion Standard Time intermediate values and have a LONG Base Type for all operations.

For the most part, Clarion's internal use of these Base Types is transparent to the programmer and do not require any consideration when planning applications. However, for business programming with numeric data containing fractional portions (currency, for instance), using data types that have the DECIMAL Base Type has some significant advantages over REAL Base Types.

- DECIMAL supports 31 significant digits of accuracy for data storage while REAL only supports 15.
- DECIMAL automatically rounds to the precision specified by the data declaration, while REAL can create rounding problems due to the translation of decimal (base 10) numbers to binary (base 2) for processing by the CPU's Floating Point Unit (or Floating Point emulation software).
- On machines without a Floating Point Unit, DECIMAL is substantially faster than REAL.
- DECIMAL operations are closely linked with conventional (decimal) arithmetic.

Type Conversion and Intermediate Results

Internally, a BCD intermediate result may have up to 31 digits of accuracy on both sides of the decimal point, so any two DECIMALs can be added with complete accuracy. Therefore, storage from BCD intermediate results to a data type can result in loss of precision. This is handled as follows :

Decimal(x,y) = BCD

First the BCD value is rounded to y decimal places. If the result overflows x digits then leading digits are removed (this corresponds to "wrapping around" a decimal counter).

Integer = BCD Any digits to the right of the decimal point are ignored. The decimal is then converted to an integer with complete accuracy and then taken modulo 2^{32} .

String(@Nx.y) = BCD

The BCD value is rounded to y decimal places, the result is fitted into the pictured string. If overflow occurs, an invalid picture (#####) results.

Real = BCD The most significant 15 digits are taken and the decimal point 'floated' accordingly.

For those operations and procedures that do not support DECIMAL types, the DECIMAL is converted to REAL first. In cases where more than 15 digits were available in the DECIMAL value, there is a loss of accuracy.

Note: Unspecified Data Type parameters have an implicit REAL Base Type, therefore DECIMAL Base Type data passed as an Unspecified Data Type Parameters will only have 15 digits of precision. DECIMAL Base Types can be passed as *DECIMAL parameters with no loss of precision.

When EVALUATEing a expression (or processing a VIEW FILTER) the REAL Base Type is used.

Simple Assignment Data Type Conversion

The rules of simple assignment data type conversion from source into destination are as follows:

BYTE =

(SHORT, USHORT, LONG, or ULONG)

The destination receives the low-order 8 bits of the source.

(DECIMAL, PDECIMAL, REAL, SREAL, BFLOAT8, or BFLOAT4)

The source is first converted to a LONG, which truncates any decimal portion, then the destination receives the low-order 8 bits of the LONG.

(STRING, CSTRING, or PSTRING)

The source must be a numeric value with no formatting characters. The source is converted to a LONG, which truncates any decimal portion, then the destination receives the low-order 8 bits of the LONG.

SHORT =

BYTE The destination receives the value of the source.

(USHORT, LONG, or ULONG)

The destination receives the low-order 16 bits of the source.

(DECIMAL, PDECIMAL, REAL, SREAL, BFLOAT8, or BFLOAT4)

The source is first converted to a LONG, which truncates any decimal portion, then the destination receives the low-order 16 bits of the LONG.

(STRING, CSTRING, or PSTRING)

The source must be a numeric value with no formatting characters. The source is first converted to a LONG, which truncates any decimal portion, then the destination receives the low-order 16 bits of the LONG.

USHORT =

BYTE The destination receives the value of the source.

(SHORT, LONG, or ULONG)

The destination receives the low-order 16 bits of the source.

(DECIMAL, PDECIMAL, REAL, SREAL, BFLOAT8, or BFLOAT4)

The source is first converted to a LONG, which truncates any decimal portion, then the destination receives the low-order 16 bits of the LONG.

(STRING, CSTRING, or PSTRING)

The source must be a numeric value with no embedded formatting characters. The source is first converted to a LONG, which truncates any decimal portion, then the destination receives the low-order 16 bits of the LONG.

LONG =

(BYTE, SHORT, USHORT, or ULONG)

The destination receives the value and the sign of the source.

(DECIMAL, PDECIMAL, REAL, SREAL, BFLOAT8, or BFLOAT4)

The destination receives the value of the source, including the sign, up to 2^{31} . If the number is greater than 2^{31} , the destination receives the result of modulo 2^{31} . Any decimal portion is truncated.

(STRING, CSTRING, or PSTRING)

The source must be a numeric value with no embedded formatting characters. The source is first converted to a REAL, which is then converted to the LONG.

DATE =**(BYTE, SHORT, USHORT, or ULONG)**

The destination receives the Btrieve format for the Clarion Standard Date for the value of the source.

(DECIMAL, PDECIMAL, REAL, SREAL, BFLOAT8, or BFLOAT4)

The source is first converted to a LONG as a Clarion Standard Date, which truncates any decimal portion, then the destination receives the Btrieve format for the Clarion Standard Date.

(STRING, CSTRING, or PSTRING)

The source must be a numeric value with no embedded formatting characters. The source is first converted to a LONG as a Clarion Standard Date, which truncates any decimal portion, then the destination receives the Btrieve format for the Clarion Standard Date.

TIME =**(BYTE, SHORT, USHORT, or ULONG)**

The destination receives the Btrieve format for the Clarion Standard Time for the value of the source.

(DECIMAL, PDECIMAL, REAL, SREAL, BFLOAT8, or BFLOAT4)

The source is first converted to a LONG as a Clarion Standard Time, which truncates any decimal portion, then the destination receives the Btrieve format for the Clarion Standard Time.

(STRING, CSTRING, PSTRING)

The source must be a numeric value with no embedded formatting characters. The source is first converted to a LONG as a Clarion Standard Time, which truncates any decimal portion, then the destination receives the Btrieve format for the Clarion Standard Time.

ULONG =**(BYTE, SHORT, or USHORT)**

The source is first converted to a LONG, then the destination receives the entire 32 bits of the LONG.

LONG The destination receives the entire 32 bits of the source.

(DECIMAL, PDECIMAL, REAL, SREAL, BFLOAT8, or BFLOAT4)

The source is first converted to a LONG, which truncates any decimal portion, then the destination receives the entire 32 bits of the LONG.

(STRING, CSTRING, or PSTRING)

The source must be a numeric value with no embedded formatting characters. The source is first converted to a LONG, which truncates any decimal portion, then the destination receives the entire 32 bits of the LONG.

REAL =**(BYTE, SHORT, USHORT, LONG, or ULONG)**

The destination receives the full integer portion and the sign of the source.

(DECIMAL, PDECIMAL, SREAL, BFLOAT8, or BFLOAT4)

The destination receives the sign, integer portion, and the decimal portion of the source.

(STRING, CSTRING, PSTRING)

The source must be a numeric string value with no embedded formatting characters. The destination receives the sign, integer, and decimal portion of the number. Trailing spaces are ignored.

SREAL =**(BYTE, SHORT, USHORT, LONG, or ULONG)**

The destination receives the sign and value of the source.

(DECIMAL, PDECIMAL, or REAL)

The destination receives the sign, integer, and fractional portion of the source.

(STRING, CSTRING, or PSTRING)

The source must be a numeric string value with no embedded formatting characters. The destination receives the sign, integer, and decimal portion of the number. Trailing spaces are ignored.

BFLOAT8 =**(BYTE, SHORT, USHORT, LONG, or ULONG)**

The destination receives the sign and value of the source.

(DECIMAL, PDECIMAL, or REAL)

The destination receives the sign, integer, and fractional portion of the source.

(STRING, CSTRING, or PSTRING)

The source must be a numeric string value with no embedded formatting characters. The destination receives the sign, integer, and decimal portion of the number. Trailing spaces are ignored.

BFLOAT4 =**(BYTE, SHORT, USHORT, LONG, or ULONG)**

The destination receives the sign and value of the source.

(DECIMAL, PDECIMAL, or REAL)

The destination receives the sign, integer, and fractional portion of the source.

(STRING, CSTRING, or PSTRING)

The source must be a numeric string value with no embedded formatting characters. The destination receives the sign, integer, and decimal portion of the number. Trailing spaces are ignored.

DECIMAL =**(BYTE, SHORT, USHORT, LONG, ULONG, or PDECIMAL)**

The destination receives the sign and the value of the source, wrapping or rounding as appropriate.

(REAL, or SREAL)

The destination receives the sign, integer, and the high order part of the fraction from the source. The high order fractional portion is rounded in the destination.

(STRING, CSTRING, PSTRING)

The source must be a numeric string value with no embedded formatting characters. The destination receives the sign, integer, and decimal portion of the number. Trailing spaces are ignored.

PDECIMAL =**(BYTE, SHORT, USHORT, LONG, ULONG, or DECIMAL)**

The destination receives the sign and the value of the source, wrapping or rounding as appropriate.

(REAL, SREAL, BFLOAT8, or BFLOAT4)

The destination receives the sign, integer, and the high order part of the fraction from the source. The high order fractional portion is rounded in the destination.

(STRING, CSTRING, or PSTRING)

The source must be a numeric string value with no embedded formatting characters. The destination receives the sign, integer, and decimal portion of the number. Trailing spaces are ignored.

STRING =**(BYTE, SHORT, USHORT, LONG, or ULONG)**

The destination receives the sign and the unformatted number. The value is left justified in the destination.

(DECIMAL, PDECIMAL, REAL, SREAL, BFLOAT8, or BFLOAT4)

The destination receives the sign, integer, and fractional portion of the source (rounded into the string's picture format). The value is left justified in the destination.

CSTRING =**(BYTE, SHORT, USHORT, LONG, or ULONG)**

The destination receives the sign and the unformatted number. The value is left justified in the destination.

(DECIMAL, PDECIMAL, REAL, SREAL, BFLOAT8, or BFLOAT4)

The destination receives the sign, integer, and fractional portion of the source (rounded into the string's picture format). The value is left justified in the destination.

PSTRING =**(BYTE, SHORT, USHORT, LONG, or ULONG)**

The destination receives the sign and the unformatted number. The value is left justified in the destination.

(DECIMAL, PDECIMAL, REAL, SREAL, BFLOAT8, or BFLOAT4)

The destination receives the sign, integer, and fractional portion of the source (rounded into the string's picture format). The value is left justified in the destination.

12 - Execution Control

Control Structures

ACCEPT (the event processor)

ACCEPT

statements

END

ACCEPT The event handler.

statements Executable code statements.

The **ACCEPT** loop is the event handler that processes events generated by Windows for the APPLICATION or WINDOW structures. An ACCEPT loop and a window are bound together, in that, when the window is opened, the next ACCEPT loop encountered will process all events for that window.

ACCEPT operates in the same manner as a LOOP--the BREAK and CYCLE statements can be used within it. The ACCEPT loop cycles for every event that requires program action. ACCEPT waits until the Clarion runtime library sends it an event that the program should process, then cycles through to execute its *statements*. During the time ACCEPT is waiting, the Clarion runtime library has control, automatically handling common events from Windows that do not need specific program action (such as screen re-draws).

The current contents of all STRING control USE variables (in the top window of each thread) automatically display on screen each time the ACCEPT loop cycles to the top. This eliminates the need to explicitly issue a DISPLAY statement to update the video display for display-only data. USE variable contents for any other control automatically display on screen for any event generated for that control, unless PROP:Auto is turned on to automatically display all USE variables each time through the ACCEPT loop.

Within the ACCEPT loop, the program determines what happened by using the following procedures:

- EVENT() Returns a value indicating what happened. Symbolic constants for events are in the EQUATES.CLW file.
- FIELD() Returns the field number for the control to which the event refers, if the event is a field-specific event.
- ACCEPTED() Returns the field number for the control to which the event refers for the EVENT:Accepted event.
- SELECTED() Returns the field number for the control to which the event refers for the EVENT:Selected event.
- FOCUS() Returns the field number of the control that has input focus, no matter what event occurred.
- MOUSEX() Returns the x-coordinate of the mouse cursor.
- MOUSEY() Returns the y-coordinate of the mouse cursor.

Two events cause an implicit BREAK from the ACCEPT loop. These are the events that signal the close of a window (EVENT:CloseWindow) or close of a program (EVENT:CloseDown). The program's code need not check for these events as they are handled automatically. However, the code may check for them and execute some specific action, such as displaying a "You sure?" window or handling some housekeeping details. A CYCLE statement at that point returns to the top of the ACCEPT loop without exiting the window or program.

Similarly, there are several other events whose action can also be terminated by a CYCLE statement: EVENT:Move, EVENT:Size, EVENT:Restore, EVENT:Maximize, and EVENT:Iconize. A CYCLE statement in response to any of these events stops the normal action and prohibits generation of the related EVENT:Moved, EVENT:Sized, EVENT:Restored, EVENT:Maximized, or EVENT:Iconized.

Example:

```
CODE
OPEN(Window)
ACCEPT                !Event handler
CASE FIELD()
OF 0                  !Handle Field-independent events
CASE EVENT()
OF EVENT:Move
  CYCLE              !Do not allow user to move the window
OF EVENT:Suspend
  CASE FOCUS()
  OF ?Field1
    !Save some stuff
  END
OF EVENT:Resume
  !Restore the stuff
END
OF ?Field1           !Handle events for Field1
CASE EVENT()
OF EVENT:Selected
  ! pre-edit code for field1
OF EVENT:Accepted
  ! completion code for field1
END
END
```

See Also:

EVENT

APPLICATION

WINDOW

FIELD

FOCUS

ACCEPTED

SELECTED

CYCLE

BREAK

CASE (selective execution structure)

```

CASE condition
OF expression [ TO expression ]
    statements
[ OROF expression [ TO expression ] ]
    statements
[ ELSE ]
    statements
END

```

CASE	Initiates a selective execution structure.
<i>condition</i>	A numeric or string variable or expression.
OF	The <i>statements</i> following an OF are executed when the <i>expression</i> following the OF option is equal to the <i>condition</i> of the CASE. There may be many OF options in a CASE structure.
<i>expression</i>	A numeric or string constant, variable, or expression.
TO	TO allows a range of values in an OF or OROF. The <i>statements</i> following the OF (or OROF) are executed if the value of the <i>condition</i> falls within the inclusive range specified by the <i>expressions</i> . The <i>expression</i> following OF (or OROF) must contain the lower limit of the range. The <i>expression</i> following TO must contain the upper limit of the range.
OROF	The <i>statements</i> following an OROF are executed when either the <i>expression</i> following the OROF or the OF option is equal to the <i>condition</i> of the CASE. There may be many OROF options associated with one OF option. An OROF may optionally be put on a separate line. An OROF does not terminate preceding <i>statements</i> groups, so control "falls into" the OROF <i>statements</i> .
ELSE	The <i>statements</i> following ELSE are executed when all preceding OF and OROF options have been evaluated as not equivalent. ELSE is not required; however, when used, it must be the last option in the CASE structure.
<i>statements</i>	Any valid Clarion executable source code.

A **CASE** structure selectively executes the first set of *statements* encountered for which there is equivalence between the *condition* and *expression* or range of *expressions*. CASE structures may be nested within other executable structures and other executable structures may be nested within CASE structures. The CASE structure must terminate with an END statement (or period).

For those situations where the program's logic could allow using either a CASE structure or a complex IF/ELSIF structure, the CASE structure will generally generate more efficient object code. EXECUTE generates the most efficient object code for those special cases where the condition evaluates to an integer in the range of 1 to n.

Example:

```

CASE ACCEPTED()                                !Evaluate field edit routine
OF ?Name                                        !If field is Name
  ERASE(?Address,?Zip)                          !erase Address through Zip
  GET(NameFile,NameKey)                         !get the record

CASE Action                                    !Evaluate Action
OF 1                                           !adding record - does not exist
  IF NOT ERRORCODE()                           !should be a file error
    ErrMsg = 'ALREADY ON FILE'                 !otherwise display error message
    DISPLAY(?Address,?Zip)                     !display address through zipcode
    SELECT(?Name)                              !re-enter the name
  END
OF 2 OROF 3                                    !change or delete - record exists
  DISPLAY(?Address,?Zip)                       !display address through zipcode
END                                             !end case action

CASE Name[1]                                   !Get first letter of name
OF 'A' TO 'M'                                  !Process first half of alphabet
OROF 'a' TO 'm'
  DO FirstHalf
OF 'N' TO 'Z' OROF 'n' TO 'z'                 !Process second half of alphabet
  DO SecondHalf
END                                             !End case sub(name)

OF ?Address                                    !If field is address
  DO AddressVal                                !call validation routine
END                                             !End case accepted()

```

See Also:

EXECUTE

IF

EXECUTE (statement execution structure)

EXECUTE *expression*

statement 1

statement 2

[**BEGIN**

statements

END]

statement n

[**ELSE]**

statement

END

EXECUTE	Initiates a single statement execution structure.
<i>expression</i>	A numeric expression or a variable that contains a numeric integer.
<i>statement 1</i>	A single statement that executes only when the <i>expression</i> is equal to 1.
<i>statement 2</i>	A single statement that executes only when the <i>expression</i> is equal to 2.
BEGIN	BEGIN marks the beginning of a structure containing a number of lines of code. The BEGIN structure will be treated as a single statement by the EXECUTE structure. The BEGIN structure is terminated by a period or the keyword END.
<i>statement n</i>	A single statement that executes only when the <i>expression</i> is equal to <i>n</i> .
ELSE	The <i>statement</i> following ELSE executes when the <i>expression</i> evaluates to a value outside the range of 1 to <i>n</i> , where <i>n</i> is defined as the total number of single statements between the EXECUTE and the ELSE.
<i>statement</i>	A single statement that executes only when the <i>expression</i> is outside the valid range.

An **EXECUTE** structure selects a single executable statement (or executable code structure) based on the value of the *expression*. The EXECUTE structure must terminate with an END statement (or period).

If the *expression* equals 1, the first statement (*statement 1*) executes. If *expression* equals 2, the second statement (*statement 2*) executes, and so on. If the value of the *expression* is zero, or greater than the total number of statements (or structures) within the EXECUTE structure, the *statement* in the ELSE clause executes. If no ELSE clause is present, program execution continues with the next statement following the EXECUTE structure.

EXECUTE structures may be nested within other executable structures and other executable code structures (IF, CASE, LOOP, EXECUTE, and BEGIN) may be nested within an EXECUTE.

For those situations where the program's logic could allow using either an EXECUTE, CASE, or an IF/ELSIF structure, the EXECUTE structure will generate more efficient object code, and is the preferred method.

Example:

```
EXECUTE Transact      !Evaluate Transact
  ADD(Customer)      !Execute if Transact = 1
  PUT(Customer)      !Execute if Transact = 2
  DELETE(Customer)   !Execute if Transact = 3
END                  !End execute

EXECUTE CHOICE()     !Evaluate CHOICE() procedure
  OrderPart          !Execute if CHOICE() = 1
  BEGIN              !Execute if CHOICE() = 2
    SavVendor" = Vendor
    UpdVendor
    IF Vendor <> SavVendor"
      Mem:Message = 'VENDOR NAME CHANGED'
    END
  END
  CASE VendorType    !Execute if CHOICE() = 3
  OF 1
    UpdPartNo1
  OF 2
    UpdPartNo2
  END
  RETURN              !Execute if CHOICE() = 4
END                  !End execute

EXECUTE SomeValue
  DO OneRoutine
  DO TwoRoutine
ELSE
  MESSAGE('SomeValue did not contain a 1 or 2')
END
```

See Also:

BEGIN

CASE

IF

IF (conditional execution structure)

```

IF logical expression [ THEN ]
    statements
[ ELSIF logical expression [ THEN ]
    statements ]
[ ELSE
    statements ]
END

```

IF Initiates a conditional statement execution structure.

logical expression

A variable, procedure, or expression which evaluates a condition. Control is determined by the result (true or false) of the expression. Zero (or blank) evaluates as false, anything else is true.

THEN The *statements* following THEN execute when the preceding *logical expression* is true. If used, THEN must only appear on the same line as **IF** or **ELSIF**.

statements An executable statement, or a sequence of executable statements.

ELSIF The *logical expression* following an **ELSIF** is evaluated only when all preceding **IF** or **ELSIF** conditions were evaluated as false.

ELSE The *statements* following **ELSE** execute only when all preceding **IF** and **ELSIF** options evaluate as false. ELSE is not required, however, when used, it must be the last option in the IF structure.

An **IF** structure controls program execution based on the outcome of one or more *logical expressions*. IF structures may have any number of ELSIF statement groups. IF structures may be "nested" within other executable structures. Other executable structures may be nested within an IF structure. Each IF structure must terminate with an END statement (or period).

Example:

```
IF Cus:TransCount = 1                                !If new customer
  AcctSetup                                           !call account setup procedure
ELSIF Cus:TransCount > 10 AND Cus:TransCount < 100 !If regular customer
  DO RegularAcct                                     !process the account
ELSIF Cus:TransCount > 100                           !If special customer
  DO SpecialAcct                                     !process the account
ELSE                                                  !Otherwise
  DO NewAcct                                         !process the account
  IF Cus:Credit
    CheckCredit
  ELSE
    CLEAR(Cus:CreditStat)
  END
  ! verify credit status
END
IF ERRORCODE()
  ErrHandler(Cus:AcctNumber,Trn:InvoiceNbr)         !Handle errors
END
```

See Also:

EXECUTE

CASE

LOOP (iteration structure)

```

label  LOOP [ | count TIMES          | ]
          | i = initial TO limit [ BY step ] |
          | UNTIL logical expression|
          | WHILE logical expression  |
          statements
      | END          |
      | UNTIL logical expression|
      | WHILE logical expression  |

```

LOOP	Initiates an iterative statement execution structure.
<i>count</i>	An integer constant, variable, or expression specifying the number of TIMES <i>statements</i> in the LOOP execute.
TIMES	Executes <i>count</i> number of iterations of the <i>statements</i> .
<i>i</i>	The label of a variable which automatically increments (or decrements, if <i>step</i> is negative) on each iteration.
= <i>initial</i>	A numeric constant, variable, or expression specifying the value of the increment variable (<i>i</i>) on the first pass through the LOOP structure.
TO <i>limit</i>	A numeric constant, variable, or expression specifying the terminating value for the LOOP . When <i>i</i> is greater than <i>limit</i> (or less than, if the <i>step</i> is a negative value) the LOOP structure control sequence terminates. The <i>i</i> variable contains the last incremental value greater than (or less than) the <i>limit</i> after the LOOP terminates.
BY <i>step</i>	A numeric constant, variable, or expression specifying the quantity by which the <i>i</i> variable increments (or decrements, if the value is negative) on each iteration of the LOOP . If BY <i>step</i> is omitted, <i>i</i> increments by 1.
UNTIL	When placed on the LOOP statement, UNTIL evaluates the <i>logical expression</i> before each iteration. When terminating the LOOP structure, UNTIL evaluates the <i>logical expression</i> after each iteration. If the <i>logical expression</i> evaluates to true, the LOOP terminates.
WHILE	When placed on the LOOP statement, WHILE evaluates the <i>logical expression</i> before each iteration. When terminating the LOOP structure, WHILE evaluates the <i>logical expression</i> after each iteration. If the <i>logical expression</i> evaluates to false, the LOOP terminates.

logical expression

A numeric or string variable, expression, or procedure. A *logical expression* evaluates a condition. Control is determined by the result (true or false) of the expression. A zero numeric or blank string value evaluates as false, anything else is true.

statements An executable statement, or a sequence of executable statements.

A **LOOP** structure repetitively executes the *statements* within its structure. LOOP structures may be nested within other executable code structures. Other executable code structures may be nested within a LOOP structure. Each LOOP structure must terminate with an END statement (or period), an UNTIL, or a WHILE statement.

A LOOP with no condition at the top or bottom iterates continuously until a BREAK or RETURN statement executes. BREAK discontinues the LOOP and continues program execution with the statement following the LOOP structure. All statements within a LOOP structure executes unless a CYCLE statement executes. CYCLE immediately sends program execution back to the top of the LOOP for the next iteration, without executing any further statements in the LOOP following the CYCLE.

LOOP UNTIL or LOOP WHILE *logical expressions* are always evaluated at the top of the LOOP, before the LOOP *statements* execute. Therefore, if the *logical expression* is false on the first pass, the LOOP *statements* will not execute even once. To create a LOOP that always executes its *statements* at least once, the UNTIL or WHILE clause must terminate the LOOP structure.

Example:

```

LOOP                                !Continuous loop
  Char = GetChar()                  !get a character
  IF Char <> CarrReturn              !if it's not a carriage return
    Field = CLIP(Field) & Char      !append the character
  ELSE                               !otherwise
    BREAK                          !break out of the loop
  END                               !End if
END                                 !end loop

IF ERRORCODE()                      !On error
  LOOP 3 TIMES                       !loop three times
    BEEP                             !sound the alarm
  END                               !End loop
END                                 !end if

LOOP I# = 1 TO 365 BY 7              !Loop, increment I# by 7 each time
  GET(DailyTotal,I#)                !read every 7th record
  DO WeeklyJob
END                                 !I# contains 372 when the LOOP terminates

```

```
LOOP I# = 10 TO 1 BY -1      !Loop, decrementing I# by 1 each time
  DO SomeRoutine
END                          !I# contains zero (0) when the LOOP terminates

SET(MasterFile)             !Point to first record
LOOP UNTIL EOF(MasterFile) !Process all the records
  NEXT(MasterFile)         !read a record
  ProcMaster               !call the procedure
END

LOOP WHILE KEYBOARD()      !Empty the keyboard buffer
  ASK                      !without processing keystrokes
UNTIL KEYCODE() = EscKey  !but break the loop for Escape
```

See Also:

BREAK

CYCLE

Execution Control Statements

BREAK (immediately leave loop)

BREAK [*label*]

BREAK Transfers control to the first statement following the terminator of a LOOP or ACCEPT structure.

label The label on the LOOP or ACCEPT statement from which to break. This must be the label of a nested loop structure containing the BREAK statement.

The **BREAK** statement immediately terminates processing in the LOOP or ACCEPT structure and transfers control to the first statement following the terminating END, WHILE, or UNTIL statement of the LOOP, or the terminating END statement of the ACCEPT structure.

BREAK may only be used in a LOOP or ACCEPT loop structure. The use of the optional *label* argument allows you to cleanly break out of multiple levels of nested loops, eliminating one common use of GOTO.

Example:

```

LOOP                                !Loop
  ASK                                !wait for a keystroke
  IF KEYCODE() = EscKey              !if Esc key pressed
    BREAK                            !break out of the loop
  ELSE                                !otherwise
    BEEP                              !sound the alarm
  END
END

Loop1 LOOP                          !Loop1 is the label
  DO ParentProcess
Loop2 LOOP                          !Loop2 is the label
  DO ChildProcess
    IF SomeCondition
      BREAK Loop1                    !Break out of both nested loops
    END
  END
END

ACCEPT                              !ACCEPT loop structure
CASE ACCEPTED()
OF ?Ok
  CallSomeProc
OF ?Cancel
  BREAK                              !break out of the loop
END;END

```

See Also: LOOP, CYCLE, ACCEPT

CYCLE (go to top of loop)**CYCLE** [*label*]**CYCLE** Transfers control back to the top of a LOOP or ACCEPT structure.*label* The label on the LOOP or ACCEPT statement to which to return. This must be the label of a nested loop structure containing the CYCLE statement.

The **CYCLE** statement passes control immediately back to the top of the LOOP or ACCEPT loop. CYCLE may only be used in a LOOP or ACCEPT loop structure. The use of the optional *label* argument allows you to cleanly go back to the top of outer levels of nested loops, eliminating one common use of GOTO.

In an ACCEPT loop, for certain events, CYCLE terminates an automatic action before it is performed. This behavior is documented for each event so affected:

- Event:Iconize
- Event:Maximize
- Event:Move
- Event:PreAlertKey
- Event:Restore
- Event:Size
- Event:Contracting
- Event:Expanding
- EVENT:CloseDown
- EVENT:CloseWindow
- EVENT:DDEPoke
- EVENT:DDEExecute

Example:

```

SET(MasterFile)           !Point to first record
LOOP                     !Process all the records
NEXT(MasterFile)         !read a record
IF ERRORCODE() THEN BREAK. !Get out of loop at end of file
DO MatchMaster           !check for a match
IF NoMatch               !if match not found
    CYCLE                 !jump to top of loop
END
DO TransVal              !validate the transaction
PUT(MasterFile)          !write the record
END

```

```
Loop1 LOOP                                !Loop1 is the label
      DO ParentProcess
Loop2 LOOP                                !Loop2 is the label
      DO ChildProcess
      IF SomeCondition
      CYCLE Loop1                          !Cycle back to top of outer loop
      END
      END
      END
```

See Also:

LOOP

BREAK

ACCEPT

DO (call a ROUTINE)

DO label

DO Executes a ROUTINE.

label The label of a ROUTINE statement.

The **DO** statement is used to execute a ROUTINE local to a PROGRAM or PROCEDURE. When a ROUTINE completes execution, program control reverts to the statement following the DO statement. A ROUTINE may only be called within the CODE section containing the ROUTINE's source code.

Example:

```
DO NextRecord      !Call the next record routine
DO CalcNetPay      !Call the calc net pay routine
```

See Also:

EXIT

ROUTINE

EXIT (leave a ROUTINE)

EXIT

The **EXIT** statement immediately leaves a ROUTINE and returns program control to the statement following the DO statement that called it. This is different from RETURN, which completely exits the PROCEDURE even when called from within a ROUTINE.

An EXIT statement is not required. A ROUTINE with no EXIT statement terminates automatically when the entire sequence of statements in the ROUTINE is complete.

Example:

```
CalcNetPay ROUTINE
  IF GrossPay = 0    !If no pay
    EXIT            ! exit the routine
  END
  NetPay = GrossPay - FedTax - Fica
  QtdNetPay += NetPay
  YtdNetPay += NetPay
```

See Also:

DO

RETURN

GOTO (go to a label)**GOTO** *target*

GOTO	Unconditionally transfers program control to another statement.
<i>target</i>	The label of another executable statement within the PROGRAM, PROCEDURE, or ROUTINE.

The **GOTO** statement unconditionally transfers control from one statement to another. The *target* of a GOTO must not be the label of a ROUTINE or PROCEDURE.

The scope of GOTO is limited to the currently executing ROUTINE or PROCEDURE--it may not *target* a label outside the ROUTINE or PROCEDURE in which it is used.

Extensive use of GOTO is generally not considered good structured programming practice. LOOP is usually considered a better alternative.

Example:

```

ComputeIt PROCEDURE(Level)
  CODE
  IF Level = 0
    GOTO PassCompute           !Skip rate calculation if no Level
  END
  Rate = Level * Markup        !Compute Rate
  RETURN(Rate)                 !and return it
PassCompute RETURN(999999)     !Return bogus number

```

See Also:

LOOP

RETURN (return to caller)**RETURN**([*expression*])**RETURN** Terminates a PROGRAM or PROCEDURE.*expression* The *expression* passes the return value of a PROCEDURE prototyped to return a value back to the expression in which the PROCEDURE was used. This may be NULL if the PROCEDURE returns a reference.

The **RETURN** statement terminates a PROGRAM or PROCEDURE and passes control back to the caller. When RETURN is executed from the CODE section of a PROGRAM, the program is terminated, all files and windows are closed, and control is passed to the operating system.

RETURN is required in a PROCEDURE prototyped to return a value and optional in a PROGRAM or PROCEDURE which does not return a value. If RETURN is not used in a PROCEDURE or PROGRAM, an implicit RETURN occurs at the end of the executable code. The end of executable code is defined as the end of the source file, or the beginning of another PROCEDURE or ROUTINE.

RETURN from a PROCEDURE (whether explicit or implicit) automatically closes any local APPLICATION, WINDOW, REPORT, or VIEW structure opened in the PROCEDURE. It does not automatically close any Global or Module Static APPLICATION, WINDOW, REPORT, or VIEW. It also closes and frees any local QUEUE structure declared without the STATIC attribute.

Example:

```

IF Done#
  RETURN                !Quit when done
END

DayOfWeek PROCEDURE(Date)    !Procedure to return the day of the week
RetVal  STRING(9)
CODE
EXECUTE Date % 7           !Determine what day of week Date is
  RetVal = 'Monday'
  RetVal = 'Tuesday'
  RetVal = 'Wednesday'
  RetVal = 'Thursday'
  RetVal = 'Friday'
  RetVal = 'Saturday'
ELSE
  RetVal = 'Sunday'
END
RETURN(RetVal)           !and RETURN the correct day string

```

See Also: PROCEDURE, PROCEDURE Return Types

13 - Built-in Procedures

Procedure Listing by Function

Logic Control

CHAIN (execute another program)
HALT (exit program)
IDLE (arm periodic procedure)
RUN (execute command)
SHUTDOWN (arm termination procedure)
STOP (suspend program execution)

Event Processing

ACCEPT (the event processor)
ALERT (set event generation key)
EVENT (return event number)
POST (post user-defined event)
REGISTER (register event handler)
UNREGISTER (unregister event handler)
YIELD (allow event processing)

Multi-Threading

START (return new execution thread)
THREAD (return current execution thread)
UNLOCKTHREAD (unlock the current execution thread)
LOCKTHREAD (re-lock the current execution thread)
THREADLOCKED (returns current execution thread locked state)
SUSPEND (suspend thread execution)
RESUME (resume thread execution)

Window Processing

ACCEPTED (return control just completed)
CHANGE (change control field value)
CHOICE (return relative item position)
CLONE (duplicate existing control)
CLOSE (close window)
CONTENTS (return contents of USE variable)
CREATE (create new control)
DESTROY (remove a control)
DISABLE (dim a control)
DISPLAY (write USE variables to screen)
ENABLE (re-activate dimmed control)
ERASE (clear screen control and USE variables)
FIELD (return control with focus)
FIRSTFIELD (return first window control)
FOCUS (return control with focus)
GETFONT (get font information)
GETPOSITION (get control position)
HELP (help window access)
HIDE (blank a control)
INCOMPLETE (return empty REQ control)
LASTFIELD (return last window control)
MESSAGE (return message box response)
MOUSEX (return mouse horizontal position)
MOUSEY (return mouse vertical position)
OPEN (open window for processing)
POPUP (return popup menu selection)
SELECT (select next control to process)
SELECTED (return control that has received focus)
SET3DLOOK (set 3D window look)
SETCURSOR (set temporary mouse cursor)
SETFONT (specify font)
SETPOSITION (specify new control position)
SETTARGET (set current window or report)
UNHIDE (show hidden control)
UPDATE (write from screen to USE variables)

Keyboard Processing

ALIAS (set alternate keycode)
ASK (get one keystroke)
FORWARDKEY (pass keystrokes to control)
KEYBOARD (return keystroke waiting)
KEYCHAR (return ASCII code)
KEYCODE (return last keycode)
KEYSTATE (return keyboard status)
PRESS (put characters in the buffer)
PRESSKEY (put a keystroke in the buffer)
SETKEYCHAR (specify ASCII code)
SETKEYCODE (specify keycode)

Windows Standard Dialogs

COLORDIALOG (return chosen color)
FILEDIALOG (return chosen file)
FONTDIALOG (return chosen font)
FONTDIALOGA (return chosen font and character set)
PRINTERDIALOG (return chosen printer)

Drag and Drop Processing

CLIPBOARD (return windows clipboard contents)
DRAGID (return matching drag-and-drop signature)
DROPID (return drag-and-drop string)
SETCLIPBOARD (set windows clipboard contents)
SETDROPID (set DROPID return string)

Non-Volatile Storage

GETINI (return INI file entry)
PUTINI (set INI file entry)
GETREG (get Windows registry entry)
PUTREG (write value to Windows registry)
DELETEREG (remove a value or key from Windows registry)

Report Processing

CLOSE (close an active report structure)
ENDPAGE (force page overflow)
OPEN (open a report structure for processing)
PRINT (print a report structure)

Graphics Processing

ARC (draw an arc of an ellipse)
BLANK (erase graphics)
BOX (draw a rectangle)
CHORD (draw a section of an ellipse)
ELLIPSE (draw an ellipse)
IMAGE (draw a graphic image)
LINE (draw a straight line)
PENCOLOR (return line draw color)
PENSTYLE (return line draw style)
PENWIDTH (return line draw thickness)
PIE (draw a pie chart)
POLYGON (draw a multi-sided figure)
ROUNDBOX (draw a box with round corners)
SETPENCOLOR (set line draw color)
SETPENSTYLE (set line draw style)
SETPENWIDTH (set line draw thickness)
SHOW (write to screen)
TYPE (write string to screen)

File Processing

BUFFER (set FILE record paging)
BUILD (build keys and indexes)
CALLBACK (register or unregister a FileCallBackInterface)
CLOSE (close a data file)
COPY (copy a file)
CREATE (create an empty data file)
EMPTY (empty a data file)
FLUSH (flush buffers)
FREESTATE (free resources)
GETSTATE (return current state of data file)
LOCK (exclusive file access)
NAME (return file name)
OPEN (open a data file)
PACK (remove deleted records)
RECORDS (return number of file or key records)
REMOVE (erase a file)
RENAME (change file directory name)
RESTORESTATE (restore state of data file)
SEND (send message to file driver)
SQLCALLBACK (register or unregister a SQLCallBackInterface)
STATUS (return file status)
STREAM (enable operating system buffering)
UNLOCK (unlock a locked data file)

Record Processing

ADD (add a new file record)
APPEND (add a new file record)
BYTES (return size in bytes)
DELETE (delete a file record)
DUPLICATE (check for duplicate key entries)
GET (read a file record by direct access)
HOLD (exclusive file record access)
NEXT (read next file record in sequence)
NOMEMO (read file record without reading memo)
POSITION (return file record sequence position)
PREVIOUS (read previous file record in sequence)
PUT (write record back to file)
RELEASE (release a held file record)
REGET (reget file record)
RESET (reset file record sequence position)
SET (initiate sequential file processing)
SKIP (bypass file records in sequence)
WATCH (automatic file concurrency check)

Transaction Processing

COMMIT (terminate successful transaction)
LOGOUT (begin transaction)
ROLLBACK (terminate unsuccessful transaction)

Null Data Processing

GETNULLS (get the NULL state of a table)
NULL (return null file field)
SETNULL (set file field null)
SETNULLS (set the null state of columns)
SETNONNULL (set file field non-null)

Internationalization Support

CONVERTANSITOOEM (convert ANSI strings to ASCII)
CONVERTOEMTOANSI (convert ASCII strings to ANSI)
ISALPHA (return alphabetic character)
ISLOWER (return lower case character)
ISUPPER (return upper case character)
LOCALE (load environment file)

View Processing

BUFFER (set VIEW record paging)
CALLBACK (register or unregister a FileCallBackInterface)
CLOSE (close a VIEW)
OPEN (open a VIEW)
DELETE (delete a view primary file record)
FLUSH (flush buffers)
HOLD (exclusive view record access)
NEXT (read next view record in sequence)
POSITION (return view record sequence position)
PREVIOUS (read previous view record in sequence)
PUT (write VIEW primary file record back)
RECORDS (return number of rows in data set)
REGET (reget view record)
RELEASE (release a held view record)
RESET (reset view record sequence position)
SET (set view record sequence position)
SKIP (bypass view records in sequence)
SQL (use SQL code)
SQLCALLBACK (register or unregister an SQLCallBackInterface)
WATCH (automatic view concurrency check)

Queue Processing

ADD (add an entry)
CHANGES (return changed queue)
DELETE (delete an entry)
FREE (delete all entries)
GET (read an entry)
POINTER (return last entry position)
POSITION (return record sequence position)
PUT (write an entry)
RECORDS (return number of entries)
SORT (sort entries)

Mathematical Procedures

ABS (return absolute value)
INRANGE (check number within range)
INT (truncate fraction)
LOGE (return natural logarithm)
LOG10 (return base 10 logarithm)
RANDOM (return random number)
ROUND (return rounded number)
SQRT (return square root)

Trigonometric Procedures

SIN (return sine)
COS (return cosine)
TAN (return tangent)
ASIN (return arcsine)
ACOS (return arccosine)
ATAN (return arctangent)

String Processing

ALL (return repeated characters)
CENTER (return centered string)
CHR (return character from ASCII)
CLIP (return string without trailing spaces)
DEFORMAT (return unformatted numbers from string)
FORMAT (return formatted numbers into a picture)
INLIST (return entry in list)
INSTRING (return substring position)
LEFT (return left justified string)
LEN (return length of string)
LOWER (return lower case)
MATCH (return matching strings)
NUMERIC (return numeric string)
RIGHT (return right justified string)
SUB (return substring of string)
TIE (associate a string value to an ASTRING)
TIED (retrieves a value associated with an ASTRING)
UNTIE (disassociate a string value from an ASTRING)
UPPER (return upper case)
VAL (return ASCII value)

Bit Manipulation

BAND (return bitwise AND)
BOR (return bitwise OR)
BXOR (return bitwise exclusive OR)
BSHIFT (return shifted bits)

Date / Time Processing

TODAY (return system date)
SETTODAY (set system date)
CLOCK (return system time)
SETCLOCK (set system time)
DATE (return standard date)
DAY (return day of month)
MONTH (return month of date)
YEAR (return year of date)
AGE (return age from base date)

Field Access

ISSTRING (return field string type or not)
WHAT (return field from group)
WHERE (return field position in group)

Operating System Procedures

COMMAND (return command line)
DIRECTORY (get file directory)
LONGPATH (return long filename)
PATH (return current directory)
RUNCODE (return program exit code)
SETCOMMAND (set command line parameters)
SETPATH (change current drive and directory)
SHORTPATH (return short filename)

Error Reporting

ERROR (return error message)
ERRORCODE (return error code number)
ERRORFILE (return error filename)
FILEERROR (return file driver error message)
FILEERRORCODE (return file driver error code number)
REJECTCODE (return reject code number)

Miscellaneous

ADDRESS (return memory address)
BEEP (sound tone on speaker)
CALL (call procedure from a DLL)
CHOOSE (return chosen value)
MAXIMUM (return maximum subscript value)
OMITTED (return omitted parameters)
PEEK (read memory address)
POKE (write to memory address)
UNLOAD (remove a CALLED DLL from memory)

ABS (return absolute value)

ABS(*expression*)

ABS Returns absolute value.

expression A constant, variable, or expression.

The **ABS** procedure returns the absolute value of an *expression*. The absolute value of a number is always positive (or zero).

Return Data Type: REAL or DECIMAL

Example:

```
C = ABS(A - B)           !C is absolute value of the difference
IF B < 0
  B = ABS(B)             !If b is negative make it positive
END
```

See Also:

BCD Operations and Procedures

ACCEPTED (return control just completed)

ACCEPTED()

The **ACCEPTED** procedure returns the field number of the control on which an EVENT:Accepted event occurred. ACCEPTED returns zero (0) for all other events.

Positive field numbers are assigned by the compiler to all WINDOW controls, in the order their declarations occur in the WINDOW structure. Negative field numbers are assigned to all APPLICATION controls. In executable code statements, field numbers are usually represented by field equate labels--the label of the USE variable preceded by a question mark (?FieldName).

Return Data Type: SIGNED

Example:

```
CASE ACCEPTED()      !Process post-edit code
OF ?Cus:Company
  !Edit field value
OF ?Cus:CustType
  !Edit field value
END
```

See Also:

ACCEPT

EVENT

ACOS (return arccosine)

ACOS(*expression*)

ACOS Returns inverse cosine.

expression A numeric constant, variable, or expression for the value of the cosine.

The **ACOS** procedure returns the inverse cosine. The inverse of a cosine is the angle that produces the cosine. The return value is the angle in radians. π is a constant which represents the ratio of the circumference and radius of a circle. There are 2π radians (or 360 degrees) in a circle.

Return Data Type: REAL

Example:

```

PI          EQUATE(3.1415926535898)      !The value of PI
Rad2Deg    EQUATE(57.295779513082)      !Number of degrees in a radian
Deg2Rad    EQUATE(0.0174532925199)      !Number of radians in a degree
CODE
  InvCosine = ACOS(CosineAngle)         !Get the Arccosine

```

See Also:

TAN

ATAN

SIN

ASIN

COS

ADD (add an entry)

```

      | file      |
      | file ,length      |
ADD(  | queue | )
      | queue, [+]key,...,[-]key |
      | queue, name      |
      | queue, function |
      | queue, pointer |

```

ADD	Writes a new record to a FILE or QUEUE.
<i>file</i>	The label of a FILE declaration.
<i>length</i>	An integer constant, variable, or expression which contains the number of bytes in the RECORD buffer to write to the <i>file</i> . If omitted or out of range, <i>length</i> defaults to the length of the RECORD structure.
<i>queue</i>	The label of a QUEUE structure, or the label of a passed QUEUE parameter.
+ -	The leading plus or minus sign specifies the <i>key</i> is sorted in ascending or descending sequence. If omitted, ascending sequence is the default.
<i>key</i>	The label of a field declared within the QUEUE structure. If the QUEUE has a PRE attribute, the <i>key</i> must include the prefix.
<i>name</i>	A string constant, variable, or expression containing the NAME attribute of QUEUE fields, separated by commas, and optional leading + or - signs for each attribute. This parameter is case sensitive.
<i>function</i>	The label of the function containing two parameters of a *GROUP or named GROUP passed by address, and having a SIGNED return value. Both parameters must use the same parameter type, and cannot be omitted. The RAW, C and PASCAL attributes are not permitted in the prototype declaration. See Additional Queue Considerations.
<i>pointer</i>	A numeric constant, variable, or numeric expression. The <i>pointer</i> must be in the range from 1 to the number of entries in the memory queue.

The **ADD** statement writes a new record to a FILE or QUEUE.

FILE Usage

All KEYS associated with the *file* are also updated during each ADD. If there is no room for the record on disk, the "Access Denied" error is posted. If an error is posted, no record is added to the file.

You can use the DUPLICATE procedure to check whether the ADD will return the "Creates Duplicate Key" error. The DUPLICATE procedure assumes that the contents of the RECORD structure data buffer are duplicated at the current record pointer location. Therefore, when using DUPLICATE prior to ADDing a record, the record pointer should be cleared with: GET(*file*,0).

ADD(*file*)

Adds a new record to the *file* by writing the entire contents of the data file's record buffer to disk.

ADD(*file,length*)

Adds a new record to the *file* by writing *length* number of bytes from the data file's record buffer to disk. The *length* must be greater than zero and not greater than the length of the RECORD. This form of ADD is not supported by all file drivers--check your file driver documentation.

QUEUE Usage

ADD writes a new entry from the QUEUE structure data buffer to the QUEUE. If there is not enough memory to ADD a new entry, the "Insufficient Memory" error is posted.

ADD(*queue*)

Appends a new entry to the end of the QUEUE.

ADD(*queue,pointer*)

Places a new entry at the relative position specified by the *pointer* parameter. If there is an entry already at the relative *pointer* position, it is "pushed down" to make room for the new entry. All following pointers are readjusted to account for the new entry. For example, an entry added at position 10 pushes entry 10 to position 11, entry 11 to position 12, etc. If *pointer* is zero or greater than the number of entries in the QUEUE, the entry is added at the end.

ADD(*queue,key*)

Inserts a new entry in a sorted memory queue. Multiple *key* parameters may be used (up to 16), separated by commas, with optional leading plus or minus signs to indicate ascending or descending sequence. The entry is inserted immediately after all other entries with matching *key* values. Using only this form of ADD will build the QUEUE in sorted order.

ADD(queue,name)

Inserts a new queue entry in a sorted memory queue. The *name* string must contain the NAME attributes of the fields, separated by commas, with optional leading plus or minus signs to indicate ascending or descending sequence. The entry is inserted immediately after all other entries with matching field values. If there are no entries, *ADD(queue,name)* may be used to build the QUEUE in sorted order.

ADD(queue,function)

Using ADD by FUNCTION will write from a positional value returned by the function. If the function returns zero (0) the queue record of the first parameter is treated as equal to the second. In this case, no record is added, since the values are equal. If the function returns a negative value, the ADD of the record passed as a first parameter is treated as having less value than record passed as second parameter and is written accordingly. If the function returns a positive value, the ADD of the record passed as a first parameter is treated as having a greater value than record passed as second parameter and is written accordingly.

If the QUEUE contains any reference variables or fields with the ANY data type, you must first CLEAR the QUEUE entry before assigning new values to the component fields of the QUEUE. This avoids possible memory leaks, since these data types automatically allocate memory.

Errors Posted: 05 Access Denied
 08 Insufficient Memory
 37 File Not Open
 40 Creates Duplicate Key
 75 Invalid Field Type Descriptor

Example:

```

NameQue QUEUE
Name      STRING(20),NAME('FirstField')
Zip       DECIMAL(5,0),NAME('SecondField')
AnyField ANY
          END

CODE
ADD(Customer)                !Add a new customer file record
IF ERRORCODE() THEN STOP(ERROR()). !and check for errors

NameQue.Name = 'Jones'      !Assign data
NameQue.Zip = 12345
NameQue.AnyField &= NEW(STRING(10)) !Create a new STRING(10) field in the QUEUE
ADD(NameQue)                !Add an entry to the end of the QUEUE
CLEAR(NameQue)              !Clear ANY for next entry

NameQue.Name = 'Taylor'    !Assign data
NameQue.Zip = 12345
NameQue.AnyField &= NEW(STRING(20)) !Create a new STRING(20) field in the QUEUE
ADD(NameQue,+NameQue.Name,-NameQue.Zip) !Ascending name, descending zip order
CLEAR(NameQue)              !Clear ANY for next entry

NameQue.Name = 'Adams'     !Assign data
NameQue.Zip = 12345
NameQue.AnyField &= NEW(STRING(30)) !Create a new STRING(30) field in the QUEUE
ADD(NameQue,1)              !Add an entry at position 1
CLEAR(NameQue)              !Clear ANY for next entry

Que:Name = 'Smith'         !Assign data
Que:Zip = 12345
NameQue.AnyField &= NEW(STRING(40)) !Create a new STRING(40) field in the QUEUE
ADD(NameQue,+FirstField,-SecondField) !Ascending name, descending zip order
CLEAR(NameQue)              !Clear ANY for next entry

```

See Also:

SORT

CLEAR

Reference Variables

PUT

GET

DUPLICATE

APPEND

ADDRESS (return memory address)

ADDRESS(| *variable* |)

ADDRESS Returns memory address of a variable.

variable The label of a data item or PROCEDURE.

The **ADDRESS** procedure returns the address of the specified data item or PROCEDURE.

The ADDRESS procedure allows you to pass the address of a *variable* or *procedure* to external libraries written in other languages, or to reference assign the address to a reference variable.

Return Data Type: LONG

Example:

```
MAP
  ClarionProc                !A Clarion language procedure
  MODULE('External.Obj')    !An external library
    ExternVarProc(LONG)     !C procedure receiving variable address
    ExternProc(LONG)        !C procedure receiving procedure address
  END
END

Var1 CSTRING(10)            !Define a null-terminated string
CODE
  ExternVarProc(ADDRESS(Var1)) !Pass address of Var1 to external procedure
  ExternProc(ADDRESS(ClarionProc)) !Pass address of ClarionProc
ClarionProc PROCEDURE      !A Clarion language procedure
CODE
  RETURN
```

See Also:

PEEK

POKE

Reference Assignment Statements

AGE (return age from base date)

AGE(*birthdate* [,*base date*])

AGE Returns elapsed time.

birthdate A numeric expression for a standard date.

base date A numeric expression for a standard date. If this parameter is omitted, the operating system date is used for the computation.

The **AGE** procedure returns a string containing the time elapsed between two dates. The age return string is in the following format:

```
1 to 60 days           - 'nn DAYS'  
61 days to 24 months - 'nn MOS'  
2 years to 999 years - 'nnn YRS'
```

Return Data Type: **STRING**

Example:

```
Message = Emp:Name & 'is ' & AGE(Emp:DOB,TODAY()) & ' old today.'
```

See Also:

Standard Date

DAY

MONTH

YEAR

TODAY

DATE

ALERT (set event generation key)

ALERT(*[first-keycode]* [, *last-keycode*])

ALERT	Specifies keys that generate an event.
<i>first-keycode</i>	A numeric keycode or keycode equate label. This may be the lower limit in a range of keycodes.
<i>last-keycode</i>	The upper limit keycode, or keycode equate label, in a range of keycodes.

ALERT specifies a key, or an inclusive range of keys, as event generation keys for the currently active window. The **ALERT** statement with no parameters clears all **ALERT** keys.

Two field-independent events, **EVENT:PreAlertKey** and **EVENT:AlertKey**, generate when the user presses the **ALERTed** key (in that order). If the code does not execute a **CYCLE** statement when processing **EVENT:PreAlertKey**, you "shortstop" the library's default action on the alerted keypress. If the code does execute **CYCLE** when processing **EVENT:PreAlertKey**, the library performs its default action for the alerted keypress. In either case, **EVENT:AlertKey** generates following **EVENT:PreAlertKey**.

Any key with a keycode may be used as the parameter of an **ALERT** statement. **ALERT** generates field-independent events, since it is not associated with any particular control. When **EVENT:AlertKey** is generated by an **ALERT** key, the **USE** variable of the control that currently has input focus is not automatically updated (use **UPDATE** if this is required). The **ALERT** statement alerts its keys separately from the **ALRT** attribute of a window or control. This means that clearing all **ALERT** keys has no effect on any keys alerted by **ALRT** attributes.

Example:

```

Screen WINDOW,ALRT(F10Key),ALRT(F9Key) !F10 and F9 alerted
      LIST,AT(109,48,50,50),USE(?List),FROM(Que),IMM
      BUTTON('&Ok'),AT(111,108,,),USE(?Ok)
      BUTTON('&Cancel'),AT(111,130,,),USE(?Cancel)
      END
CODE
OPEN(Screen)
ALERT                                !Turn off all alerted keys
ALERT(F1Key,F12Key)                 !Alert all function keys
ALERT(279)                           !Alert the Ctrl-Esc key
ACCEPT
CASE EVENT()
OF EVENT:PreAlertKey                !Pre-check alert events
  IF KEYCODE() <> F4Key              !Dis-Allow F4 key standard library action, and allow
    CYCLE                            !all other F keys to perform their standard functions
  END
OF EVENT:AlertKey                   !Alert processing
CASE KEYCODE()
OF 279                               !Check for Ctrl+Esc
  BREAK
OF F9Key                             !Check for F9
  F9HotKeyProc                       !Call hot key procedure
OF F10Key                            !Check for F10
  F10HotKeyProc                      !Call hot key procedure
END
END
END
END

```

ALIAS (set alternate keycode)

ALIAS([*keycode*,*[new keycode]*])

ALIAS	Changes the keycode generated when the original key is pressed.
<i>keycode</i>	A numeric keycode or keycode EQUATE. If both parameters are omitted, all ALIASed keys are reset to their original values.
<i>new keycode</i>	A numeric keycode or keycode EQUATE. If omitted, the <i>keycode</i> is reset to its original value.

ALIAS changes the *keycode* to generate the *new keycode* when the user presses the original key. ALIAS does not affect keypresses generated by PRESSKEY. The effect of ALIAS is global, throughout all execution threads, no matter where the ALIAS statement executes. Therefore, to only change the *keycode* locally, you must reset ALIASed keys when the window loses focus.

Keycode values 0800h through 0FFFFh are unassigned and may be used as a *new keycode*. The practical effect of this is to disable the original key if your program does not test for the *new keycode*.

Example:

```
ALIAS(EnterKey,TabKey)  !Allow user to press enter instead of tab
ALIAS(F3Key,F1Key)     !Move help to F3
ALIAS                   !Clear all aliased keys
```

See Also:

KEYCODE

ALL (return repeated characters)

ALL(*string* [,*length*])

ALL	Returns repeated characters.
<i>string</i>	A string expression containing the character sequence to be repeated.
<i>length</i>	The length of the return string. If omitted the <i>length</i> of the return string is 255 characters.

The **ALL** procedure returns a string containing repetitions of the character sequence *string*.

Return Data Type: **STRING**

Example:

```
Starline = ALL('*',25)      !Get 25 asterisks
Dotline  = ALL('.')        !Get 255 dots
```

APPEND (add a new file record)

APPEND(*file* [,*length*])

APPEND	Writes a new record to a FILE.
<i>file</i>	The label of a FILE declaration.
<i>length</i>	An integer constant, variable, or expression which contains the number of bytes to write to the <i>file</i> . The <i>length</i> must be greater than zero and not greater than the length of the RECORD. If omitted or out of range, <i>length</i> defaults to the length of the RECORD structure.

The **APPEND** statement writes a new record from the RECORD structure data buffer to the data file. No KEYS associated with the *file* are updated during an APPEND. After APPENDING records, the KEYS must be rebuilt with the BUILD command.

APPEND is usually used in batch processes, to speed the process of adding a large number of records at one time to the *file*. For most every file system, it is much faster to add 5000 records to a *file* using APPEND (and then issue BUILD at the end of the process to rebuild all the keys at once) than it is to use ADD to add the same 5000 records (which automatically updates the keys with each new record added).

If an error is posted, no record is added to the file. If there is no room for the record on disk, the "Access Denied" error is posted.

Errors Posted: 05 Access Denied
 37 File Not Open

Example:

```

LOOP                               !Process an input file
NEXT(InFile)                       !getting each record in turn
IF ERRORCODE()                    !break loop on error
  BREAK
END
Cus:Record = Inf:Record            !Copy the data to Customer file
APPEND(Customer)                  !and APPEND a customer record
IF ERRORCODE()                    !check for errors
  STOP(ERROR())
END
END
BUILD(Customer)                   !Re-build Keys

```

See Also:

BUILD, ADD

ARC (draw an arc of an ellipse)

ARC(*x* , *y* , *width* , *height* , *startangle* , *endangle* [, *attributelist*])

ARC	Draws an arc of an ellipse on the current window or report.
<i>x</i>	An integer expression that specifies the horizontal position of the starting point.
<i>y</i>	An integer expression that specifies the vertical position of the starting point.
<i>width</i>	An integer expression that specifies the width.
<i>height</i>	An integer expression that specifies the height.
<i>startangle</i>	An integer expression that specifies the starting point of the arc, in tenths of degrees (10 = 1 degree) measured counter-clockwise from three o'clock.
<i>endangle</i>	An integer expression that specifies the ending point of the arc, in tenths of degrees (10 = 1 degree) measured counter-clockwise from three o'clock.
<i>attributelist</i>	A string constant, variable, or EQUATE containing an optional type of output document and its associated attributes. Only valid when the target is a REPORT. (See <i>EXTEND</i>)

The **ARC** procedure places an arc of an ellipse on the current target. The ellipse is drawn inside a "bounding box" defined by the *x*, *y*, *width*, and *height* parameters. The *x* and *y* parameters specify the starting point, and the *width* and *height* parameters specify the horizontal and vertical size of the "bounding box."

The *startangle* and *endangle* parameters specify what sector of the ellipse will be drawn, as an arc.

The border color is the current pen color set by **SETPENCOLOR**; the default color is the Windows system color for window text. The border width is the current width set by **SETPENWIDTH**; the default width is one pixel. The border style is the current pen style set by **SETPENSTYLE**; the default style is a solid line.

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
    !window controls
    END
CODE
OPEN(MDIChild)
ARC(100,50,100,50,0,900)    !Draw 90 degree arc from 3 to 12 o'clock, as
                           !the top-right quadrant of ellipse
```

See Also:

Current Target,SETPENCOLOR,SETPENWIDTH,SETPENSTYLE

ASK (get one keystroke)

ASK

ASK reads a single keystroke from the keyboard buffer. Program execution stops to wait for a keystroke. If there is already a keystroke in the keyboard buffer, ASK gets one keystroke without waiting. The ASK statement also allows any TIMER attribute events to generate and cycle their own ACCEPT loop. This means any batch processing code can allow other threads to execute their TIMER attribute tasks during the batch process.

Example:

```
ASK                !Wait for a keystroke
LOOP WHILE KEYBOARD() !Empty the keyboard buffer
  ASK              !without processing keystrokes
END
```

See Also:

KEYCODE

KEYBOARD

ASIN (return arcsine)

ASIN(*expression*)

ASIN Returns inverse sine.

expression A numeric constant, variable, or expression for the value of the sine.

The **ASIN** procedure returns the inverse sine. The inverse of a sine is the angle that produces the sine. The return value is the angle in radians. π is a constant which represents the ratio of the circumference and radius of a circle. There are 2π radians (or 360 degrees) in a circle.

Return Data Type: REAL

Example:

```

PI           EQUATE(3.1415926535898)      !The value of PI
Rad2Deg      EQUATE(57.295779513082)     !Number of degrees in a radian
Deg2Rad      EQUATE(0.0174532925199)     !Number of radians in a degree
CODE
  InvSine = ASIN(SineAngle)              !Get the Arcsine

```

See Also:

TAN

ATAN

SIN

COS

ACOS

ATAN (return arctangent)

ATAN(*expression*)

ATAN Returns inverse tangent.

expression A numeric constant, variable, or expression for the value of the tangent.

The **ATAN** procedure returns the inverse tangent. The inverse of a tangent is the angle that produces the tangent. The return value is the angle in radians. π is a constant which represents the ratio of the circumference and radius of a circle. There are 2π radians (or 360 degrees) in a circle.

Return Data Type REAL

Example:

```

PI           EQUATE(3.1415926535898)      !The value of PI
Rad2Deg     EQUATE(57.295779513082)      !Number of degrees in a radian
Deg2Rad     EQUATE(0.0174532925199)      !Number of radians in a degree
CODE
  InvTangent = ATAN(TangentAngle)        !Get the Arctangent

```

See Also:

TAN

SIN

ASIN

COS

ACOS

BAND (return bitwise AND)

BAND(*value*,*mask*)

BAND	Performs bitwise AND operation.
<i>value</i>	A numeric constant, variable, or expression for the bit <i>value</i> to be compared to the bit <i>mask</i> . The <i>value</i> is converted to a LONG data type prior to the operation, if necessary.
<i>mask</i>	A numeric constant, variable, or expression for the bit <i>mask</i> . The <i>mask</i> is converted to a LONG data type prior to the operation, if necessary.

The **BAND** procedure compares the *value* to the *mask*, performing a Boolean AND operation on each bit. The return value is a LONG integer with a one (1) in the bit positions where the *value* and the *mask* both contain one (1), and zeroes in all other bit positions.

BAND is usually used to determine whether an individual bit, or multiple bits, are on (1) or off (0) within a variable.

Return Data Type: LONG

Example:

```
!BAND(0110b,0010b) returns 0010b !0110b = 6, 0010b = 2
```

```
RateType  BYTE                !Type of rate
Female    EQUATE(0001b)       !Female mask
Male      EQUATE(0010b)       !Male mask
Over25    EQUATE(0100b)       !Over age 25 mask
```

```
CODE
IF BAND(RateType,Female) |    !If female
  AND BAND(RateType,Over25)  ! and over 25
  DO BaseRate                !use base premium
ELSIF BAND(RateType,Male)    !If male
  DO AdjBase                 !adjust base premium
END
```

See Also:

BOR

BXOR

BSHIFT

BEEP (sound tone on speaker)

BEEP(*[sound]*)

BEEP	Generates a sound through the system speaker.
<i>sound</i>	A numeric constant, variable, expression, or EQUATE for the Windows sound to issue.

The **BEEP** statement generates a sound through the system speaker. These are standard Windows sounds available through the [sounds] section of the WIN.INI file. Standard EQUATE values similar to these are listed in the EQUATES.CLW file:

```
BEEP:SystemDefault
BEEP:SystemHand
BEEP:SystemQuestion
BEEP:SystemExclamation
BEEP:SystemAsterisk
```

Example:

```
IF ERRORCODE()           !If unexpected error
  BEEP(BEEP:SystemDefault) !sound a standard beep
  STOP(ERROR())           !stop for the error
END
```

BLANK (erase graphics)

BLANK([*x*] [,*y*] [,*width*] [,*height*])

BLANK	Erases all graphics written to the specified area of the current window or report.
<i>x</i>	An integer expression that specifies the horizontal position of the starting point. If omitted, the default is zero.
<i>y</i>	An integer expression that specifies the vertical position of the starting point. If omitted, the default is zero.
<i>width</i>	An integer expression that specifies the width. If omitted, the default is the width of the window.
<i>height</i>	An integer expression that specifies the height. If omitted, the default is the height of the window.

The **BLANK** procedure erases all graphics written to the specified area of the current window or report. Controls are not erased. BLANK with no parameters erases the entire window or report.

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
    !window controls
    END
CODE
OPEN(MDIChild)
ARC(100,50,100,50,0,900) !Draw arc
BLANK                    !Then erase it
```

See Also:

Current Target

BOR (return bitwise OR)

BOR(*value*,*mask*)

BOR	Performs bitwise OR operation.
<i>value</i>	A numeric constant, variable, or expression for the bit <i>value</i> to be compared to the bit <i>mask</i> . The <i>value</i> is converted to a LONG data type prior to the operation, if necessary.
<i>mask</i>	A numeric constant, variable, or expression for the bit <i>mask</i> . The <i>mask</i> is converted to a LONG data type prior to the operation, if necessary.

The **BOR** procedure compares the *value* to the *mask*, performing a Boolean OR operation on each bit. The return value is a LONG integer with a one (1) in the bit positions where the *value*, or the *mask*, or both, contain a one (1), and zeroes in all other bit positions.

BOR is usually used to unconditionally turn on (set to one), an individual bit, or multiple bits, within a variable.

Return Data Type: LONG

Example:

```
!BOR(0110b,0010b) returns 0110b !0110b = 6, 0010b = 2
```

```
RateType  BYTE                !Type of rate
Female    EQUATE(0001b)       !Female mask
Male      EQUATE(0010b)       !Male mask
Over25    EQUATE(0100b)       !Over age 25 mask
CODE
RateType = BOR(RateType,Over25) !Turn on over 25 bit
RateType = BOR(RateType,Male)   !Set rate to male
```

See Also:

BAND

BXOR

BSHIFT

BOX (draw a rectangle)

BOX(*x* , *y* , *width* , *height* [, *fill*] [, *attributelist*])

BOX	Draws a rectangular box on the current window or report.
<i>x</i>	An integer expression that specifies the horizontal position of the starting point.
<i>y</i>	An integer expression that specifies the vertical position of the starting point.
<i>width</i>	An integer expression that specifies the width.
<i>height</i>	An integer expression that specifies the height.
<i>fill</i>	A LONG integer constant, constant EQUATE, or variable containing the red, green, and blue components that create the color in the three low-order bytes (bytes 0, 1, and 2) or an EQUATE for a standard Windows color value.
<i>attributelist</i>	A string constant, variable, or EQUATE containing an optional type of output document and its associated attributes. Only valid when the target is a REPORT. (See <i>EXTEND</i>)

The **BOX** procedure places a rectangular box on the current window or report. The position and size of the box are specified by *x*, *y*, *width*, and *height* parameters.

The *x* and *y* parameters specify the starting point, and the *width* and *height* parameters specify the horizontal and vertical size of the box. The box extends to the right and down from its starting point.

The border color is the current pen color set by **SETPENCOLOR**; the default color is the Windows system color for window text. The border width is the current width set by **SETPENWIDTH**; the default width is one pixel. The border style is the current pen style set by **SETPENSTYLE**; the default style is a solid line.

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
    !window controls
    END
CODE
OPEN(MDIChild)
BOX(100,50,100,50,00FF0000h)    !Red box
```

See Also:

Current Target

SETPENCOLOR, SETPENWIDTH, SETPENSTYLE

BSHIFT (return shifted bits)

BSHIFT(*value,count*)

BSHIFT	Performs the bit shift operation.
<i>value</i>	A numeric constant, variable, or expression. The <i>value</i> is converted to a LONG data type prior to the operation, if necessary.
<i>count</i>	A numeric constant, variable, or expression for the number of bit positions to be shifted. If <i>count</i> is positive, <i>value</i> is shifted left. If <i>count</i> is negative, <i>value</i> is shifted right.

The **BSHIFT** procedure shifts a bit *value* by a bit *count*. The bit value may be shifted left (toward the high order), or right (toward the low order). Zero bits are supplied to fill vacated bit positions when shifting.

Return Data Type: LONG

Example:

```
!BSHIFT(0110b,1)  returns 1100b
!BSHIFT(0110b,-1) returns 0011b
```

```
Varswitch = BSHIFT(20,3)      !Multiply by eight
Varswitch = BSHIFT(Varswitch,-2) !Divide by four
```

See Also:

BAND

BOR

BXOR

BUFFER (set record paging)

BUFFER(*entity* [, *pagesize*] [, *behind*] [, *ahead*] [, *timeout*])

BUFFER	Specifies FILE or VIEW paging.
<i>entity</i>	The label of a FILE or VIEW structure.
<i>pagesize</i>	An integer constant or variable which specifies the number of records in a single "page" of records (PROP:FetchSize). If omitted, the default value is one (1).
<i>behind</i>	An integer constant or variable which specifies the number of "pages" of records to store after they've been read. If omitted, the default value is zero (0).
<i>ahead</i>	An integer constant or variable which specifies the number of additional "pages" of records to read ahead of the currently displayed page. If omitted, the default value is zero (0).
<i>timeout</i>	An integer constant or variable which specifies the number of seconds the buffered records are considered not to be obsolete in a network environment. If omitted, the default value is zero (0), which indicates no time limit.

The **BUFFER** statement specifies automatic record set buffering for the specified *entity* by the file driver. If there are multiple file drivers used by the files in a VIEW *entity*, BUFFER is ignored.

The number of records in a single "page" of records is specified by the *pagesize* parameter. This is also the number of records fetched in a single call to the database. The *ahead* parameter specifies asynchronous read-ahead buffering of a number of pages, while the *behind* parameter saves pages of already read records.

The records in the buffer must be contiguous. Therefore, issuing a SET to an area of the *entity* that is not currently in the buffer, or changing the sort order or the FILTER condition of a VIEW, will clear the buffers. The buffers remain active until the *entity* is closed, or a FLUSH statement is issued. The buffers will reflect the results of ADD, PUT, or DELETE statements, however, this may cause an implicit flush if a PUT changes key components or an ADD adds a record that is not within the current contiguous set of buffered records.

BUFFER allows the performance of "browse" type procedures to be virtually instantaneous when displaying pages of records already read, due to use of the *ahead* and *behind* parameters. BUFFER can also optimize performance when the file driver is a Client/Server back-end database engine (usually SQL-based), since the file driver can then optimize the calls made to the back-end database for minimum network traffic.

BUFFER is not supported by all file drivers--see the relevant file driver's documentation for further information.

Example:

```
CODE
```

```
OPEN(MyView)
```

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

```
CODE
```

```
OPEN(MyFile)
```

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

See Also: FLUSH,

BUILD (build keys and indexes)

```

      | file |
BUILD( |index | [, components [, filter ] ] )
      | key |

```

BUILD	Builds keys and indexes.
<i>file</i>	The label of a FILE declaration.
<i>index</i>	The label of an INDEX declaration.
<i>key</i>	The label of a KEY declaration.
<i>components</i>	A string constant or variable containing the list of the component fields on which to BUILD the dynamic INDEX. The fields must be separated by commas, with leading plus (+) or minus (-) to indicate ascending or descending sequence (if supported by the file driver).
<i>filter</i>	A string constant, variable, or expression containing a logical expression with which to filter out unneeded records from the dynamic <i>index</i> . This requires that you name <i>components</i> for the <i>index</i> . You must BIND all variables used in the <i>filter</i> expression.

The **BUILD** statement re-builds keys and indexes in a FILE..

BUILD(*file*)

Builds all the KEYS declared for the file. The file must be closed, LOCKed, or opened with *access mode* set to 12h (ReadWrite/DenyAll) or 22h (ReadWrite/DenyWrite).

BUILD(*key*) or BUILD(*index*)

Builds only the specified KEY or INDEX. The file must be closed, LOCKed, or opened with *access mode* set to either 12h (ReadWrite/DenyAll) or 22h (ReadWrite/DenyWrite).

BUILD(*index,components,filter*)

Builds a dynamic INDEX. This form does not require exclusive access to the file, however, the file must be open (with any valid *access mode*). The dynamic INDEX is created as a temporary file, exclusive to the user who BUILDS it. The temporary file is automatically deleted when the file is closed. If a *filter* is specified, the resulting INDEX will contain only those records which meet the *filter* criteria. The *filter* must be in a format supported by the file driver.

BUILD will generate events to the currently open window if you assign a value (an integer from 1 to 100) to PROP:ProgressEvents for the affected FILE before you issue the BUILD. The larger the value you assign to PROP:ProgressEvents, the more events are generated and the slower the BUILD will progress. These events allow you to indicate to the user the progress of the BUILD. This can keep end-users informed that BUILD is still working while building large files (so they don't re-boot thinking the machine has locked up).

It is not valid to make any calls to the *file* being built except to query its properties, call NAME(*file*), or CLOSE(*file*) (which aborts the process and is not recommended). Issuing a CYCLE statement in response to any of the events generated (except EVENT:BuildDone) cancels the operation. During the BUILD operation, *file*{PROP:Completed} returns the percentage completed of the re-build and you can use *file*{PROP:CurrentKey} to get a key reference then either *key*{PROP:Name} or *key*{PROP:Label} to return the name of the current key being built.

Errors Posted: 37 File Not Open
 40 Creates Duplicate Key
 63 Exclusive Access Required
 76 Invalid Index String
 93 BUILD Cancelled

Events Generated:

EVENT:BuildFile	BUILD(<i>file</i>) is rebuilding the data portion of the <i>file</i> .
EVENT:BuildKey	BUILD(<i>key</i>) or BUILD(<i>index</i>) is rebuilding the key, or BUILD(<i>file</i>) is rebuilding the keys in the <i>file</i> .
EVENT:BuildDone	The BUILD is complete. If the user cancelled the BUILD, ERRORCODE 93 is set.

Example:

```

Names    FILE,DRIVER('TopSpeed'),PRE(Nam)  !Declare a file structure
NameKey  KEY(Nam:Name),OPT                 !Declare name key
NbrNdx   INDEX(Nam:Number),OPT             !Declare number index
DynNdx   INDEX()                           !Declare a dynamic index
Rec      RECORD
Name     STRING(20),NAME('Nam:Name')
Number   SHORT,NAME('Nam:Number')
        END
        END

CODE
OPEN(Names,12h)                            !Open file, exclusive read/write
BUILD(Names)                               !Build all keys on Names file
BUILD(Nam:NbrNdx)                           !Build the number index

!Build dynamic index ascending number, ascending name:
BUILD(Nam:DynNdx,'+Nam:Number,+Nam:Name')

BIND('Nam:Name',Nam:Name)                  !BIND the filter variable

!Build dynamic index of names that start with A:
BUILD(Nam:DynNdx,'+Nam:Name','UPPER(Nam:Name[1]) = A')

UNBIND('Nam:Name')                          !UNBIND the filter variable

```

See Also:

OPEN

SHARE

BIND

PROP:ProgressEvents

PROP:Completed

BXOR (return bitwise exclusive OR)

BXOR(*value*,*mask*)

BXOR	Performs bitwise exclusive OR operation.
<i>value</i>	A numeric constant, variable, or expression for the bit <i>value</i> to be compared to the bit <i>mask</i> . The <i>value</i> is converted to a LONG data type prior to the operation, if necessary.
<i>mask</i>	A numeric constant, variable, or expression for the bit <i>mask</i> . The <i>mask</i> is converted to a LONG data type prior to the operation, if necessary.

The **BXOR** procedure compares the *value* to the *mask*, performing a Boolean XOR operation on each bit. The return value is a LONG integer with a one (1) in the bit positions where either the *value* or the *mask* contain a one (1), but not both. Zeroes are returned in all bit positions where the bits in the *value* and *mask* are alike.

BXOR is usually used to toggle on (1) or off (0) an individual bit, or multiple bits, within a variable.

Return Data Type: LONG

Example:

```
!BXOR(0110b,0010b) returns 0100b  !0110b = 6, 0100b = 4, 0010b = 2
```

```
RateType  BYTE           !Type of rate
Female    EQUATE(0001b)  !Female mask
Male      EQUATE(0010b)  !Male mask
Over25    EQUATE(0100b)  !Over age 25 mask
Over65    EQUATE(1100b)  !Over age 65 mask
CODE
RateType = BXOR(RateType,Over65)  !Toggle over 65 bits
```

See Also:

BAND

BOR

BSHIFT

BYTES (return size in bytes)

BYTES(*file*)

BYTES Returns number of bytes in FILE, or most recently read.

file The label of a FILE.

The **BYTES** procedure returns the size of a FILE in bytes or the number of bytes in the last record successfully accessed. Following an OPEN statement, BYTES returns the size of the file. After the *file* has been successfully accessed by GET, REGET, NEXT, PREVIOUS, ADD, or PUT, the BYTES procedure returns the number of bytes accessed in the RECORD. The BYTES procedure may be used to return the number of bytes read in a variable length record.

Return Data Type: LONG

Example:

```

DosFileName  STRING(260),STATIC
LastRec      LONG
SavPtr       LONG(1)           !Start at 1
FileSize     LONG

DosFile      FILE,DRIVER('DOS'),PRE(DOS),NAME(DosFileName)
Record       RECORD
F1           STRING(2000)
            END
            END

BlobStorage  FILE,DRIVER('TopSpeed'),PRE(STO)
File         BLOB,BINARY
Record       RECORD
FileName     STRING(64)
            END
            END

CODE
IF NOT FILEDIALOG('Choose File to Store',DosFileName,,0010b)
  RETURN
END
OPEN(BlobStorage)           !Open the BLOB file
STO:FileName = DosFileName  !and store the filename
OPEN(DosFile)              !Open the file
FileSize = BYTES(DosFile)   !Get size of file
STO:File{PROP:Size} = FileSize !and set the BLOB to store the file
LastRec = FileSize % SIZE(DOS:Record) !Check for short record at end of file
LOOP INT(FileSize/SIZE(DOS:Record)) TIMES

```

```
GET(DosFile,SavPtr)           !Get each record
ASSERT(NOT ERRORCODE())
!String slice data into BLOB:
STO:File[SavPtr - 1 : SavPtr + SIZE(DOS:Record) - 2] = DOS:Record
SavPtr += SIZE(DOS:Record)    !Compute next record pointer
END

IF LastRec                    !If short record at end of file
GET(DosFile,SavPtr)          !Get last record
ASSERT(BYTES(DosFile) = LastRec) ! size read should match computed size
STO:File[SavPtr - 1 : SavPtr + LastRec - 2] = DOS:Record
END
ADD(BlobStorage)
ASSERT(NOT ERRORCODE())
CLOSE(DosFile);CLOSE(BlobStorage)
```

See Also:

OPEN

CALL (call procedure from a DLL)

CALL(*file*, *procedure* [, *flags*])

CALL	Calls a procedure that has not been prototyped in the application's MAP structure from a Windows standard .DLL.
<i>file</i>	A string constant, variable, or expression containing the name (including extension) of the .DLL to open. This may include a full path.
<i>procedure</i>	A string constant, variable, or expression containing the name of the <i>procedure</i> to call (which may not receive parameters or return a value). This can also be the ordinal number indicating the <i>procedure's</i> position within the .DLL.
<i>flags</i>	An UNSIGNED integer constant, variable, or expression containing bitmap flag settings.

The **CALL** procedure calls a *procedure* from a Windows-standard .DLL. The *procedure* does not need to be prototyped in the application's MAP structure. If it is not already loaded by Windows, the .DLL *file* is loaded into memory. The .DLL *file* is automatically unloaded from memory when the *procedure* terminates unless the lowest *flags* bit is set to one (1). A .DLL *file* left loaded may be explicitly unloaded with the UNLOAD procedure.

CALL returns zero (0) for a successful *procedure* call. If unsuccessful, it can return one of the following mapped error values, or any other valid Windows level error code:

- 1 Procedure name cannot be resolved in a specified .DLL
- 2 File not found
- 3 Path not found
- 5 Attempted to load a task, not a .DLL
- 6 Library requires separate data segments for each task
- 10 Wrong Windows version
- 11 Invalid .EXE file (DOS file or error in program header)
- 12 OS/2 application
- 13 DOS 4.0 application
- 14 Unknown .EXE type
- 15 Attempt to load an .EXE created for an earlier version of Windows.
- 16 Attempt to load a second instance of an .EXE file containing multiple,writeable data segments.
- 17 EMS memory error on the second loading of a .DLL
- 18 Attempt to load a protected-mode-only application while Windows is running in Real mode

Return Data Type: SIGNED

Example:

```
X# = CALL('CUSTOM.DLL','1') !Call first procedure in CUSTOM.DLL
IF X# THEN STOP(X#).       !Check for successful execution
```

See Also: UNLOAD

CALLBACK (register or unregister a FileCallbackInterface)

CALLBACK(*entity*, *FileCallbackInterface*, [*flag*])

CALLBACK Register or unregister a FileCallbackInterface.

entity The label of a FILE or VIEW.

FileCallbackInterface

The label of the interface that implements the FileCallbackInterface. The methods of the FileCallbackInterface are called automatically before (method FunctionCalled) and after (method FunctionDone) each file operation.

flag An integer constant, variable, EQUATE, or expression that indicates whether or not to unregister an interface associated with a FILE or VIEW. A value of one (1 or TRUE) unregisters the interface. If omitted, the interface is registered with the *entity*.

The Callback method registers a callback interface with the specified *entity*. The methods of the registered interface are called whenever a file operation is done. Multiple interfaces can be registered with an entity.

To unregister an interface, set the *flag* to TRUE. Any registered interfaces must be unregistered before the object that implements the interface is removed.

Example:

```

PROGRAM

    MAP
    END
    INCLUDE ('FILECB.INC'),ONCE

!Data file
People      FILE,DRIVER('TOPSPEED'),PRE(PEO),CREATE,BINDABLE,THREAD
KeyId       KEY(PEO:Id),NOCASE,OPT
KeyLastName KEY(PEO:LastName),DUP,NOCASE
Record      RECORD,PRE()
Id          LONG
FirstName   STRING(30)
LastName    STRING(30)
Gender      LONG
            END
            END

!Log File
LogFile     FILE,DRIVER('BASIC','/ALWAYSQUOTE=OFF /COMMA=1,1'),CREATE,NAME('logfile.txt')
Record      RECORD
Operation   STRING(200)
            END
            END

!FileCallback Class
FCB CLASS,IMPLEMENTS(FileCallbackInterface)
    END

    CODE
    CALLBACK(People, FCB.FileCallbackInterface) !Register FCB interface
    CREATE(Logfile)                             !Create log file
    OPEN(Logfile)                               !Open log file
    OPEN(People)                                !Open data file
    SET(PEO:KeyId, PEO:KeyID)                   !Set and
    LOOP                                         !loop thru
        NEXT(People)                             !data until
        IF ERRORCODE()
            BREAK                                 !end of file
        END
    END
    CLOSE(People)                               !Close data file

!Unregister FCB interface:
CALLBACK(People, FCB.FileCallbackInterface, TRUE)

```

!This method is called prior to each operation of the data file.
!The log file is updated with the file operation that is being executed.

```
FCB.FileCallBackInterface.FunctionCalled |
    PROCEDURE(SIGNED opCode, *Params Parameters, *CSTRING ErrCode, *CSTRING ErrMsg)
p LIKE(Params)
CODE
    p = Parameters
    CASE opCode
    OF DriverOp:ADD
        logFile.Operation = 'ADD(f)'
    OF DriverOp:APPEND
        logFile.Operation = 'APPEND(f)'
    OF DriverOp:CLOSE
        logFile.Operation = 'CLOSE(f)'
    OF DriverOp:COPY
        logFile.Operation = 'COPY(f, ''&CLIP(Parameters.Text)&'')'
    OF DriverOp:CREATE
        logFile.Operation = 'CREATE(f)'
    OF DriverOp:DELETE
        logFile.Operation = 'DELETE(f)'
    OF DriverOp:NEXT
        logFile.Operation = 'NEXT(f)'
    OF DriverOp:OPEN
        logFile.Operation = 'OPEN(f, '&Parameters.openMode&')'
    OF DriverOp:PUT
        logFile.Operation = 'PUT(f)'
    OF DriverOp:SETkeykey
        logFile.Operation = 'SET(k,k)'
    END
    ADD(logFile)
    RETURN TRUE
```

!This method is called after each operation to the data file.
!This simply returns a TRUE according to the rules of the FileCallBackInterface.

```
FCB.FileCallBackInterface.FunctionDone |
    PROCEDURE(SIGNED opCode, Params Parameters, *CSTRING ErrCode, *CSTRING ErrMsg)
CODE
    RETURN TRUE
```

CENTER (return centered string)

CENTER(*string* [,*length*])

CENTER	Returns centered string.
<i>string</i>	A string constant, variable or expression.
<i>length</i>	The length of the return string. If omitted, the length of the <i>string</i> parameter is used.

The **CENTER** procedure first removes leading and trailing spaces from a *string*, then pads it with leading and trailing spaces to center it within the *length*, and returns a centered string.

Return Data Type: **STRING**

Example:

```
!CENTER('ABC',5) returns ' ABC '
!CENTER('ABC ') returns ' ABC '
!CENTER(' ABC') returns ' ABC '
```

```
Message = CENTER(Message)    !Center the message
Rpt:Title = CENTER(Name,60)  !Center the name
```

See Also:

LEFT

RIGHT

CHAIN (execute another program)

CHAIN(*program*)

CHAIN Terminates the current program and executes another.

program A string constant or variable containing the name of the program to execute. This may be any .EXE or .COM program.

CHAIN terminates the current program, closing all files and returning its memory to the operating system, and executes another *program*.

Example:

```
PROGRAM                !MainMenu program code
CODE
EXECUTE CHOICE()
  CHAIN('Ledger')      !Execute LEDGER.EXE
  CHAIN('Payroll')     !Execute PAYROLL.EXE
  RETURN               !Return to DOS
END

PROGRAM                !Ledger program code
CODE
EXECUTE CHOICE()
  CHAIN('MainMenu')    !Return to MainMenu program
  RETURN               !Return to DOS
END

PROGRAM                !Payroll program code
CODE
EXECUTE CHOICE()
  CHAIN('MainMenu')    !Return to MainMenu program
  RETURN               !Return to DOS
END
```

See Also:

RUN

CHANGE (change control field value)

CHANGE(*control,value*)

CHANGE Changes the *value* displayed in a *control* in an APPLICATION or WINDOW structure.

control Field number or field equate label of a window control field.

value A constant or variable containing the *control's* new value.

The **CHANGE** statement changes the *value* displayed in a *control* in an APPLICATION or WINDOW structure. CHANGE updates the *control's* USE variable with the *value*, and then displays that new *value* in the control field.

Example:

```
Screen WINDOW,PRE(Scr)
    ENTRY(@N3),USE(Ctl:Code)
    ENTRY(@S30),USE(Ctl:Name)
    BUTTON('OK'),USE(?OkButton),KEY(EnterKey)
    BUTTON('Cancel'),USE(?CanxButton),KEY(EscKey)
END

CODE
OPEN(Screen)
ACCEPT
CASE EVENT()
OF EVENT:Selected
CASE SELECTED()
OF ?Ctl:Code
CHANGE(?Ctl:Code,4) !Change Ctl:Code to 4 and display it
OF ?Ctl:Name
CHANGE(?Ctl:Name,'ABC Company') !Change Ctl:Name to ABC Company and display
END
OF EVENT:Accepted
CASE ACCEPTED()
OF ?OkButton
BREAK
OF ?CanxButton
CLEAR(Ctl:Record)
BREAK
END
END
```

See Also:

DISPLAY

UPDATE

CHANGES (return changed queue)

CHANGES(*queue*)

CHANGES Returns a "hash" value for the QUEUE.

queue The label of a QUEUE structure, or the label of a passed QUEUE parameter.

The **CHANGES** procedure returns a LONG integer containing a unique "hash" value for the current QUEUE contents. Saving this value then later comparing the saved value to the current return value from CHANGES allows you to easily detect that the contents of the QUEUE have changed (in any way at all).

Return Data Type: LONG

Example:

```
SaveHash LONG
Que       QUEUE
Name     STRING(10)
        END

CODE
Que.Name = 'Jones'
ADD(Que)                !Add the entry
ASSERT(~ERRORCODE())
SaveHash = CHANGES(Que) !Save the "hash" value
Que.Name = 'Jones II'
ADD(Que)                !Add another entry
ASSERT(~ERRORCODE())
IF SaveHash <> CHANGES(Que) !This should be a true expression here
    MESSAGE('CHANGES procedure worked correctly')
END
```

See Also:

QUEUE

CHOICE (return relative item position)

CHOICE([*control*])

CHOICE Returns a user selection number.

control A field equate label of a LIST, COMBO, SHEET, or OPTION control.

The **CHOICE** procedure returns the sequence number of a selected item in an OPTION structure, SHEET structure, LIST box, or COMBO control. With no parameter, CHOICE returns the sequence number of the selected item in the last control (LIST, SHEET, OPTION, or COMBO) that generated a Field-specific event to cycle the ACCEPT loop. CHOICE(*control*) returns the current selection number of any LIST, SHEET, OPTION, or COMBO in the currently active window.

CHOICE returns the sequence number of the selected RADIO control within an OPTION structure. The sequence number is determined by relative position within the OPTION. The first control listed in the OPTION structure's code is relative position 1, the second is 2, etc.

CHOICE returns the memory QUEUE entry number of the selected item when a LIST or COMBO box is completed.

Return Data Type: SIGNED

Example:

```

CODE
ACCEPT
  EXECUTE CHOICE() !Perform menu option
  AddRec          !procedure to add record
  PutRec          !procedure to change record
  DelRec          !procedure to delete record
  RETURN          !return to caller
END
END

```

See Also:

LIST

SHEET

COMBO

OPTION

QUEUE

RADIO

CHOOSE (return chosen value)

```
CHOOSE(      | expression ,value, value [,value...] | )
           | condition [,value, value ] |
```

CHOOSE	Returns the chosen value from a list of possible values.
<i>expression</i>	An arithmetic expression which determines which <i>value</i> parameter to return. This expression must resolve to a positive integer.
<i>value</i>	A variable, constant, or expression for the procedure to return.
<i>condition</i>	A logical expression which determines which of the two <i>value</i> parameters to return. If no <i>value</i> parameters are present, one (1) is returned when the <i>expression</i> is true, and zero (0) is returned when the <i>expression</i> is false.

The **CHOOSE** procedure evaluates the *expression* or *condition* and returns the appropriate *value* parameter. If the *expression* resolves to a positive integer, that integer selects the corresponding *value* parameter for the CHOOSE procedure to return. If the *expression* evaluates to an out-of-range integer, then CHOOSE returns the last *value* parameter.

When the *condition* evaluates as true, then CHOOSE returns the first *value* parameter. When the *condition* evaluates to false, then CHOOSE returns the second *value* parameter. If no *value* parameters are present, CHOOSE returns one (1) for true, and zero (0) for false.

The return data type is dependent upon the data types of the *value* parameters:

<u>All Value Parameters</u>	<u>Return Data Type</u>
LONG	LONG
DECIMAL or LONG	DECIMAL
STRING	STRING
DECIMAL, LONG, or STRING	DECIMAL
anything else	REAL

Return Data Type: LONG, DECIMAL, STRING, or REAL

Example:

```
!CHOOSE(4,'A','B','C','D','E') returns 'D'  
!CHOOSE(1 > 2,'A','B')       returns 'B'  
!CHOOSE(1 > 2)                 returns zero (0)
```

!Hide or unhide control, based on the value in SomeField:

```
?MyControl{PROP:Hide} = CHOOSE(SomeField = 0,TRUE,FALSE)
```

!VIEW filter to select "overweight" people of both sexes

```
MyView{PROP:Filter} = 'Weight > CHOOSE(Sex = 'M',250,200)'
```

See Also:

INLIST

CHORD (draw a section of an ellipse)

CHORD(*x* , *y* , *width* , *height* , *startangle* , *endangle* [, *fill*] [, *attributelist*])

CHORD	Draws a closed sector of an ellipse on the current window or report.
<i>x</i>	An integer expression that specifies the horizontal position of the starting point.
<i>y</i>	An integer expression that specifies the vertical position of the starting point.
<i>width</i>	An integer expression that specifies the width.
<i>height</i>	An integer expression that specifies the height.
<i>startangle</i>	An integer expression that specifies the starting point of the chord, in tenths of degrees (10 = 1 degree) measured counter-clockwise from three o'clock.
<i>endangle</i>	An integer expression that specifies the ending point of the chord, in tenths of degrees (10 = 1 degree) measured counter-clockwise from three o'clock.
<i>fill</i>	A LONG integer constant, constant EQUATE, or variable containing the red, green, and blue components that create the color in the three low-order bytes (bytes 0, 1, and 2) or an EQUATE for a standard Windows color value.
<i>attributelist</i>	A string constant, variable, or EQUATE containing an optional type of output document and its associated attributes. Only valid when the target is a REPORT. (See <i>EXTEND</i>)

The **CHORD** procedure places a closed sector of an ellipse on the current window or report. The ellipse is drawn inside a "bounding box" defined by the *x*, *y*, *width*, and *height* parameters. The *x* and *y* parameters specify the starting point, and the *width* and *height* parameters specify the horizontal and vertical size of the "bounding box." The *startangle* and *endangle* parameters specify what sector of the ellipse will be drawn, as an arc. The two end points of the arc are also connected with a straight line.

The border color is the current pen color set by **SETPENCOLOR**; the default color is the Windows system color for window text. The border width is the current width set by **SETPENWIDTH**; the default width is one pixel. The border style is the current pen style set by **SETPENSTYLE**; the default style is a solid line.

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
    !window controls
    END
CODE
OPEN(MDIChild)
CHORD(100,50,100,50,0,900,00FF0000h)    !Red 90 degree crescent
```

See Also: Current Target, **SETPENCOLOR**, **SETPENWIDTH**, **SETPENSTYLE**

CHR (return character from ASCII)

CHR(*code*)

CHR Returns the display character.

code A numeric expression containing a numeric ASCII character code.

The **CHR** procedure returns the ANSI character represented by the ASCII character *code* parameter.

Return Data Type: **STRING**

Example:

```
Stringvar = CHR(122)    !Get lower case z  
Stringvar = CHR(65)    !Get upper case A
```

See Also:

VAL

CLIP (return string without trailing spaces)

CLIP(*string*)

CLIP Removes trailing spaces.

string A string expression.

The **CLIP** procedure removes trailing spaces from a *string*. The return string is a substring with no trailing spaces. CLIP is frequently used with the concatenation operator in string expressions using STRING data types.

CLIP is not normally needed with CSTRING data types, since these have a terminating character. CLIP is also not normally needed with PSTRING data types, since these have a length byte.

When used in conjunction with the LEFT procedure, you can remove both leading and trailing spaces (frequently called ALLTRIM in other languages).

Return Data Type: **STRING**

Example:

```
Name = CLIP>Last) & ', ' & CLIP(First) & Init & '.'    !Full name in military order
```

```
Name = CLIP(First) & CLIP(' ' & Middle) & ' ' & Last !Full name with or with middle
```

```
AllTrimVar = CLIP(LEFT(MyVar))                            !Trim leading and trailing spaces
```

See Also:

LEFT

CLIPBOARD (return windows clipboard contents)

CLIPBOARD([*format*])

CLIPBOARD Returns the current contents of the Windows clipboard.

format An integer constant or variable that defines the format of the clipboard's contents. If omitted, the default is CF_TEXT .

The **CLIPBOARD** procedure returns the current contents of the windows clipboard. The *format* parameter defaults to CF_TEXT (as defined in the Windows API) but any of the other CF_ values can be specified (see a Windows API reference book for details). If the data in the clipboard is not in the specified *format*, CLIPBOARD returns a null string ("). The following clipboard formats are predefined in the Windows API:

```
CF_TEXT 1
CF_BITMAP 2
CF_METAFILEPICT 3
CF_SYLK 4
CF_DIF 5
CF_TIFF 6
CF_OEMTEXT 7
CF_DIB 8
CF_PALETTE 9
CF_PENDATA 10
CF_RIFF 11
CF_WAVE 12
```

Return Data Type: STRING

Example:

```
Que1 QUEUE
    STRING(30)
    END
Que2 QUEUE
    STRING(30)
    END
WinOne WINDOW,AT(0,0,160,400)
    LIST,AT(120,0,20,20),USE(?List1),FROM(Que1),DRAGID('List1')
    LIST,AT(120,120,20,20),USE(?List2),FROM(Que2),DROPID('List1','~FILE')
    END
CODE
OPEN(WinOne)
ACCEPT
CASE EVENT()
OF EVENT:Drag                !When a drag event is attempted
    IF DRAGID()                !check for success
        SETCLIPBOARD(Que1)    !and setup info to pass
    END
OF EVENT:Drop                !When drop event is successful
    Que2 = CLIPBOARD()        !get dropped info
    ADD(Que2)                  !and add it to the queue
END
END
```

See Also:

SETCLIPBOARD

CLOCK (return system time)

CLOCK()

The **CLOCK** procedure returns the time of day from the operating system time in standard time (expressed as hundredths of a second since midnight, plus one). Although the time is expressed to the nearest hundredth of a second, the system clock is only updated 18.2 times a second (approximately every 5.5 hundredths of a second).

Return Data Type: **LONG**

Example:

```
Time = CLOCK()      !Save the system time
```

See Also:

Standard Time

SETCLOCK

CLONE(duplicate existing control)

CLONE(*destination control*, *source control* [,*parent*] [,*position*] [,*window*])

CLONE Duplicates an existing control.

destination control

A field number or field equate label for the control to create.

source control A field number or field equate label for the control to duplicate.

parent A field number or field equate label that specifies the OPTION, GROUP, SHEET, TAB, MENU, HEADER, FOOTER, DETAIL, BREAK, or FORM to contain the new *control*. If omitted, the control has no *parent*.

position An integer constant, expression, or variable that specifies the position within a MENU to place a new ITEM *control*. If omitted, the ITEM is added to the end.

window The label of an APPLICATION, WINDOW, or REPORT structure, or a reference to any of those structures where the control to duplicate exists.

CLONE dynamically duplicates an existing control to the currently active window. It returns the field equate label of the new control. The destination control inherits all properties of the source control. The position of the destination control should be modified using SETPOSITION() or PROP:AT if it will not be positioned on top of the source control.

CLONE may only be used to duplicate controls. It may not be used to duplicate report bands, menu items, or OLE controls.

Return Data Type: SIGNED

Example:

PROGRAM

```
INCLUDE('EQUATES.CLW')
INCLUDE('KEYCODES.CLW')
```

```
MAP
END
```

```
WINDOW WINDOW('Clone - example'),AT(,,260,100),GRAY
    BUTTON('Ok'),AT(198,30,31,15),USE(?OkButton)
    BUTTON('Clone It'),AT(199,54,31,15),USE(?CloneButton)
END
```

```
WINDOW2 WINDOW('Window2'),AT(,,260,100),GRAY
    BUTTON('Clone Me'),AT(98,30,31,15),USE(?CloneMe)
END
```

```
?Cloned          EQUATE(100)
CODE
OPEN(WINDOW)
ACCEPT
CASE ACCEPTED()
  OF ?CloneButton
    OPEN(WINDOW2)
    SETTARGET(WINDOW)
    CLONE(?Cloned, ?CloneMe,,,WINDOW2)
    CLOSE(WINDOW2)
    ?Cloned{PROP:YPOS} = ?Cloned{PROP:Ypos} + ?Cloned{PROP:Height} +10
    ?Cloned{PROP:Width} = ?Cloned{PROP:Width} + 20
    ?Cloned{PROP:Text} = 'Cloned Button'
  OF ?OkButton
    BREAK
END
END
```

CLOSE (close a data structure)

CLOSE(*entity*)

CLOSE Closes a data structure.

entity The label of a FILE, VIEW, APPLICATION, WINDOW, or REPORT structure.

CLOSE terminates processing on the active *entity*. Any memory used by the active *entity* is released when it is closed.

FILE Usage

CLOSE(file) closes an active FILE. Generally, this flushes DOS buffers and frees any memory used by the open file other than the RECORD structure's data buffer. If the file is a member of a transaction set, error 56 (LOGOUT active) is posted.

VIEW Usage

CLOSE(view) closes an active VIEW. A VIEW declared within a procedure is implicitly closed upon RETURN from the procedure, if it has not already been explicitly CLOSED. If the CLOSE(view) statement is not immediately preceded by a REGET statement, the state of the primary and secondary related files in the VIEW are undefined. The contents of their record buffers are undefined and a SET or RESET statement may be required before performing sequential processing on the file.

APPLICATION and WINDOW Usage

CLOSE(window) closes an active APPLICATION or WINDOW structure. Memory used by the active window is released when it is closed and the underlying screen is automatically re-drawn. When a window is closed, if it is not the top-most window on its execution thread, all windows opened subsequent to the window being closed are automatically closed first. This occurs in the reverse order from which they were opened. An APPLICATION or WINDOW that is declared local to (within) a PROCEDURE is automatically closed when the program RETURNS from the procedure.

REPORT Usage

CLOSE(report) closes an active REPORT structure, which prints the last page FOOTER (unless the last structure printed has the ALONE attribute) and any required group FOOTER structures, and closes the REPORT. If the REPORT has the PREVIEW attribute, all the temporary metafiles are deleted. RETURN from a procedure in which a REPORT is opened automatically closes the REPORT.

Errors Posted: 56 LOGOUT active

Example:

```

Customer      FILE,DRIVER('Clarion'),PRE(Cus) !Declare customer file layout
AcctKey       KEY(Cus:AcctNumber)
Record        RECORD
AcctNumber    LONG
OrderNumber   LONG
Name          STRING(20)
Addr          STRING(20)
City          STRING(20)
State         STRING(20)
Zip           STRING(20)
              END
              END

ViewCust      VIEW(Customer)                !Declare VIEW structure
              PROJECT(Cus:AcctNumber,Cus:Name)
              END

CODE
OPEN(Customer,22h)
SET(Cus:AcctKey)
OPEN(ViewCust)                               !Open the customer view
!executable statements
CLOSE(ViewCust)                               !and close it again

CLOSE(Customer)                               !Close the customer file
CLOSE(MenuScr)                                !Close the menu screen
CLOSE(CustEntry)                             !Close customer data entry screen
CLOSE(CustRpt)                               !Close the report

```

See Also:

OPEN

LOGOUT

ACCEPT

COLORDIALOG (return chosen color)

COLORDIALOG([*title*] ,*rgb* [, *suppress*])

COLORDIALOG	Displays the Windows standard color choice dialog box to allow the user to choose a color.
<i>title</i>	A string constant or variable containing the title to place on the color choice dialog. If omitted, a default <i>title</i> is supplied by Windows.
<i>rgb</i>	A LONG integer variable to receive the selected color.
<i>suppress</i>	An integer constant or variable containing either zero (0) or one (1). If one, the list of standard colors is suppressed. If omitted or zero (0) the list of standard colors is displayed.

The **COLORDIALOG** procedure displays the Windows standard color choice dialog box and returns the color chosen by the user in the *rgb* parameter. Any existing value in the *rgb* parameter sets the default color choice presented to the user in the color choice dialog. The color chosen by the user may be either an RGB value (a positive value) or one of the Windows standard element colors (a negative value).

COLORDIALOG returns zero (0) if the user pressed the Cancel button, or one (1) if the user pressed the Ok button on the color choice dialog.

Return Data Type: SIGNED

Example:

```
MDIChild1 WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
    !window controls
END
```

ColorNow LONG

```
CODE
IF NOT COLORDIALOG('Choose Box Color',ColorNow)
    ColorNow = 000000FFh          !Default to Red if user pressed Cancel
END
OPEN(MDIChild1)
BOX(100,50,100,50,ColorNow)    !User-defined color for box
```

See Also:

COLOR

FONT

COMMAND (return command line)

COMMAND([*flag*])

COMMAND Returns command line parameters.

flag A string constant or variable containing the parameter for which to search, or the number of the command line parameter to return. If omitted or an empty string (""), all command parameters are returned as entered on the command line, appended to a leading space.

The **COMMAND** procedure returns the value of the *flag* parameter from the command line. If the *flag* is not found, **COMMAND** returns an empty string. If the *flag* is multiply defined, only the first occurrence encountered is returned.

COMMAND searches the command line for *flag=value* and returns *value*. There must be no blanks between *flag*, the equal sign, and *value*. The returned *value* terminates at the first comma or blank space. If a blank or comma is desired in a command line parameter, everything to the right of the equal sign must be enclosed in double quotes (*flag="value"*).

COMMAND will also search the command line for a *flag* containing a leading slash (/). If found, **COMMAND** returns the value of *flag* without the slash. If the *flag* only contains a number, **COMMAND** returns the parameter at that numbered position on the command line. A *flag* of '0' returns the minimum path the operating system used to find the command. This minimum path always includes the command (without command line parameters) but may not include the path (if the operating system found it in the current directory). A *flag* containing '1' returns the first command line parameter.

Return Data Type: **STRING**

Example:

```

IF COMMAND('/N')                !Was /N on the command line?
  DO SomeProcess
END
IF COMMAND('Option') = '1'     !Was Option=1 on the command line?
  DO OneProcess
END
CommandString = COMMAND('')    !Get all command parameters
CommandItself = COMMAND('0')   !Get the command itself
SecondParm = COMMAND('2')      !Get second parameter from command line

```

See Also:

SETCOMMAND

COMMIT (terminate successful transaction)

COMMIT

The **COMMIT** statement terminates an active transaction. Execution of a COMMIT statement assumes that the transaction was completely successful and no ROLLBACK is necessary. Once COMMIT has been executed, ROLLBACK of the transaction is impossible.

COMMIT informs the file driver involved in the transaction that the temporary files containing the information necessary to restore the database to its previous state may be deleted. The file driver then performs the actions necessary to its file system to successfully terminate a transaction.

```
Errors Posted:    48    Unable to Log Transaction
                  91    No Logout Active
```

Example:

```
LOGOUT(1,OrderHeader,OrderDetail)  !Begin Transaction
DO ErrorHandler                    !always check for errors
ADD(OrderHeader)                    !Add Parent record
DO ErrorHandler                    !always check for errors
LOOP X# = 1 TO RECORDS(DetailQue)  !Process stored detail records
  GET(DetailQue,X#)                !Get one from the QUEUE
  DO ErrorHandler                    !always check for errors
  Det:Record = DetailQue            !Assign to record buffer
  ADD(OrderDetail)                  !and add it to the file
  DO ErrorHandler                    !always check for errors
END
COMMIT                              !Terminate successful transaction
ASSERT(~ERRORCODE())

ErrorHandler ROUTINE                !Error routine
IF NOT ERRORCODE() THEN EXIT.       !Bail out if no error
Err" = ERROR()                      !Save the error message
ROLLBACK                            !Rollback the aborted transaction
ASSERT(~ERRORCODE())
BEEP                                 !Alert the user
MESSAGE('Transaction Error - ' & Err)
RETURN                               !and get out
```

See Also:

LOGOUT

ROLLBACK

CONTENTS (return contents of USE variable)

CONTENTS(*control*)

CONTENTS Returns the value in the USE variable of a control.

control A field number or field equate label.

The **CONTENTS** procedure returns a string containing the value in the USE variable of an ENTRY, OPTION RADIO, or TEXT control.

A USE variable may be longer than its associated control field display picture OR may contain fewer characters than its total capacity. The CONTENTS procedure always returns the full length of the USE variable.

Return Data Type: **STRING**

Example:

```
IF CONTENTS(?LastName)='' AND CONTENTS(?FirstName)='' !If first and last name blank,  
  MessageField = 'Must Enter a First or Last Name' ! display error message  
END
```

CONVERTANSITOOEM (convert ANSI strings to ASCII)

CONVERTANSITOOEM(*string*)

CONVERTANSITOOEM

Translates ANSI strings to OEM ASCII.

string The label of the string to convert. This may be a single variable or a any structure that is treated as a GROUP (RECORD, QUEUE, etc.).

The **CONVERTANSITOOEM** statement translates either a single string or the strings within a GROUP from the ANSI (Windows display) character set into the OEM character set (ASCII with extra characters defined by the active code page).

This procedure is not required on data files if the OEM attribute is set on the file.

Example:

```
Customer  FILE,DRIVER('Clarion'),PRE(Cus)  !Declare file without OEM attribute
AcctKey   KEY(Cus:AcctNumber)
Record    RECORD
AcctNumber LONG
OrderNumber LONG
Name      STRING(20)
Addr      STRING(20)
          END
          END

Win WINDOW,SYSTEM
    STRING(@s20),USE(Cus:Name)
    END

CODE
OPEN(Customer)
SET(Customer)
NEXT(Customer)
CONVERTOEMTOANSI(Cus:Record)           !Convert all strings from ASCII to
ANSI
OPEN(Win)
ACCEPT
    !Process window controls
    END
CONVERTANSITOOEM(Cus:Record)          !Convert back to ASCII from ANSI
PUT(Customer)
```

See Also: CONVERTOEMTOANSI, OEM

CONVERTOEMTOANSI (convert ASCII strings to ANSI)

CONVERTOEMTOANSI(*string*)

CONVERTOEMTOANSI

Translates OEM ASCII strings to ANSI.

string The label of the string to convert. This may be a single variable or a any structure that is treated as a GROUP (RECORD, QUEUE, etc.).

The **CONVERTOEMTOANSI** statement translates either a single string or the strings within a GROUP from the the OEM character set (ASCII with extra characters defined by the active code page) into ANSI (Windows display) character set.

This procedure is not required on data files if the OEM attribute is set on the file.

Example:

```
Customer  FILE,DRIVER('Clarion'),PRE(Cus)  !Declare file without OEM attribute
AcctKey   KEY(Cus:AcctNumber)
Record    RECORD
AcctNumber LONG
OrderNumber LONG
Name      STRING(20)
Addr      STRING(20)
          END
          END

Win WINDOW,SYSTEM
    STRING(@s20),USE(Cus:Name)
    END

CODE
OPEN(Customer)
SET(Customer)
NEXT(Customer)
CONVERTOEMTOANSI(Cus:Record)           !Convert all strings from ASCII to ANSI
OPEN(Win)
ACCEPT
    !Process window controls
    END
CONVERTANSITOOEM(Cus:Record)          !Convert back to ASCII from ANSI
PUT(Customer)
```

See Also: CONVERTANSITOOEM, OEM

COPY (copy a file)

COPY(*file,new file*)

COPY	Duplicates a file.
<i>file</i>	The label of a FILE structure, or a string constant or variable containing the file specification of the file to copy.
<i>new file</i>	A string constant or variable containing a file specification. If the file specification does not contain a drive and path, the current drive and directory are assumed. If only the path is specified, the filename and extension of the original <i>file</i> are used for the <i>new file</i> .

The **COPY** statement duplicates a FILE and enters the specification for the *new file* in the operating system's directory listing. The *file* to copy must be closed, or the "File Already Open" error is posted. If the file specification of the *new file* is identical to the original *file*, the COPY statement is ignored.

Since some file drivers use multiple physical disk files for one logical FILE structure, the default filename and extension assumptions are dependent on the file driver. If any error is posted, the file is not copied.

Errors Posted: 02 File Not Found
 03 Path Not Found
 05 Access Denied
 52 File Already Open

Example:

```
TheFile  STRING(256),STATIC
SomeFile FILE,DRIVER('DOS'),NAME(TheFile)
Record   RECORD
F1       STRING(1)
         END
         END

CODE
TheFile = 'Names.DAT'
COPY(TheFile,'A:')           !Copy file to floppy
COPY('C:\AUTOEXEC.BAT','A:\AUTOEXEC.BAT') !Copy file to floppy
```

See Also:

CLOSE

COS (return cosine)

COS(*radians*)

COS Returns cosine.

radians A numeric constant, variable or expression for the angle in radians. p is a constant which represents the ratio of the circumference and radius of a circle. There are $2p$ radians (or 360 degrees) in a circle.

The **COS** procedure returns the trigonometric cosine of an angle measured in *radians*. The cosine is the ratio of the length of the angle's adjacent side divided by the length of the hypotenuse.

Return Data Type: REAL

Example:

```

PI          EQUATE(3.1415926535898)  !The approx. value of PI
Rad2Deg     EQUATE(57.295779513082)  !Number of degrees in a radian
Deg2Rad     EQUATE(0.0174532925199)  !Number of radians in a degree
CODE
  Angle = 45 * Deg2Rad                !Translate 45 degrees to Radians
  CosineAngle = COS(Angle)            !Get the cosine of 45 degree angle

```

See Also:

TAN

ATAN

SIN

ASIN

ACOS

CREATE (create an empty data file)

CREATE(*file*)

CREATE Creates an empty data file.

file The label of the FILE to be created.

The **CREATE** statement adds an empty data file to the operating system directory. If the *file* already exists, it is deleted and recreated as an empty file. The *file* must be closed, or the "File Already Open" error is posted. CREATE does not open the file for access.

Errors Posted: 03 Path Not Found
 04 Too Many Open Files
 05 Access Denied
 52 File Already Open
 54 No Create Attribute

Example:

```
CREATE(Master)     !Create a new master file
CREATE(Detail)    !Create a new detail file
```

See Also:

CLOSE

CREATE (return new control created)

CREATE(*control* , *type* [, *parent*] [, *position*])

CREATE	Creates a new control.
<i>control</i>	A field number or field equate label for the control to create. The valid range of field numbers is -4000h to 4000h. If the control equal to 0, the CREATE procedure returns the next available field number and assigns that to the control being created.
<i>type</i>	An integer constant, expression, EQUATE, or variable that specifies the type of control to create.
<i>parent</i>	A field number or field equate label that specifies the OPTION, GROUP, SHEET, TAB, MENU, HEADER, FOOTER, DETAIL, BREAK, or FORM to contain the new <i>control</i> . If omitted, the control has no <i>parent</i> .
<i>position</i>	An integer constant, expression, or variable that specifies the position within a MENU to place a new ITEM <i>control</i> . If omitted, the ITEM is added to the end.

CREATE dynamically creates a new control in the currently active APPLICATION or WINDOW, returning the value of the *control* parameter. If the field creation fails, a 0 is returned.

When first created, the new *control* is initially hidden, so its properties can be set using the runtime property assignment syntax, SETPOSITION, and SETFONT. It appears on screen only by issuing a UNHIDE statement for the *control*.

You can also use CREATE to create report controls. To do this, you must first use SETTARGET to make the report the currently active TARGET, and you must also specify a *parent* for the control.

EQUATE statements for the *type* parameter are contained in the EQUATES.CLW file. The following list is a comprehensive sample of these (see EQUATES.CLW for the complete list):

CREATE:sstring	STRING(picture),USE(variable)
CREATE:string	STRING(constant)
CREATE:image	IMAGE()
CREATE:region	REGION()
CREATE:line	LINE()
CREATE:box	BOX()
CREATE:ellipse	ELLIPSE()
CREATE:entry	ENTRY()
CREATE:button	BUTTON()
CREATE:prompt	PROMPT()
CREATE:option	OPTION()
CREATE:radio	RADIO()

CREATE:check	CHECK()
CREATE:group	GROUP()
CREATE:list	LIST()
CREATE:combo	COMBO()
CREATE:spin	SPIN()
CREATE:text	TEXT()
CREATE:custom	CUSTOM()
CREATE:droplist	LIST(),DROP()
CREATE:dropcombo	COMBO(),DROP()
CREATE:menu	MENU()
CREATE:item	ITEM()

Return Data Type: SIGNED

Example:

PROGRAM

```
INCLUDE('keycodes.clw')
```

```
MAP
```

```
END
```

```
TestGroup GROUP,PRE(CTL)
```

```
CODE LONG
```

```
Name STRING(30)
```

```
END
```

```
X SHORT
```

```
Y SHORT
```

```
Width SHORT
```

```
Height SHORT
```

```
Code4Entry STRING(10)
```

```
?Code4Entry EQUATE(100) !Create an arbitrary field equate number for CREATE  
to use
```

```
FEQ EQUATE(101)
```

```
UseVarText1 &STRING
```

```
Window WINDOW,VSCROLL,GRAY,MAXIMIZE
```

```
ENTRY(@N3),AT(3,32),USE(Ctl:Code)
```

```
ENTRY(@S30),AT(98,8),USE(Ctl:Name)
```

```
BUTTON('OK'),AT(5,7),USE(?OkButton)
```

```
BUTTON('Cancel'),AT(45,7),USE(?CanxButton)
```

```
END
```

```

CODE
OPEN(Window)
ACCEPT
CASE ACCEPTED()
OF ?Ctl:Code
  IF Ctl:Code = 4
    CREATE(?Code4Entry,CREATE:entry)           !Create the control
    ?Code4Entry{PROP:use} = Code4Entry         !Set USE variable
    ?Code4Entry{PROP:text} = '@s10'           !Set entry picture
    GETPOSITION(?Ctl:Code,X,Y,Width,Height)
    ?Code4Entry{PROP:Xpos} = X + Width + 40    !Set x position
    ?Code4Entry{PROP:Ypos} = Y                 !Set y position
    UNHIDE(?Code4Entry)                       !Display the new control
  ELSIF Ctl:Code = 5
    CREATE(FEQ, CREATE:text)
    UseVarText1 &= NEW(STRING(10000))
    FEQ{PROP:use} = UseVarText1
    GETPOSITION(?Ctl:Code,X,Y,Width,Height)
    FEQ{PROP:Xpos} = X + Width + 40           !Set x position
    FEQ{PROP:Ypos} = Y + 5                   !Set y position
    FEQ{PROP:width} = 300
    FEQ{PROP:height} = 100
    FEQ{PROP:Vscroll} = TRUE
    UNHIDE(FEQ)                               !Display the new control
  END
OF ?OkButton
  BREAK
OF ?CanxButton
  CLEAR(TestGroup)
  BREAK
END
END
CLOSE(Window)

RETURN

```

See Also:

DESTROY

SETTARGET

DATE (return standard date)

DATE(*month,day,year*)

DATE	Return standard date.
<i>month</i>	A positive numeric constant, variable, or expression for the <i>month</i> .
<i>day</i>	A positive numeric constant, variable, or expression for the <i>day</i> of the month.
<i>year</i>	A numeric constant, variable or expression for the <i>year</i> . The valid range for a <i>year</i> value is 00 through 99 (using "Intellidate" logic), or 1801 through 2099.

The **DATE** procedure returns a standard date for a given *month*, *day*, and *year*. The *month* and *day* parameters do allow positive out-of-range values (zero or negative values are invalid). A *month* value of 13 is interpreted as January of the next year. A *day* value of 32 in January is interpreted as the first of February. Consequently, DATE(12,32,97), DATE(13,1,97), and DATE(1,1,98) all produce the same result.

The century for a two-digit *year* parameter is resolved using the default "Intellidate" logic, which assumes the date falls in the range of the next 20 or previous 80 years from the current operating system date. For example, assuming the current year is 1998, if the *year* parameter is "15," the date returned is in the year 2015, and if the *year* parameter is "60," the date returned is in 1960.

Return Data Type: LONG

Example:

```
HireDate = DATE(Hir:Month,Hir:Day,Hir:Year)           !Compute hire date
FirstOfMonth = DATE(MONTH(TODAY()),1,YEAR(TODAY())) !Compute First day of month
```

See Also:

Standard Date

DAY

MONTH

YEAR

TODAY

DAY (return day of month)

DAY(*date*)

DAY	Returns day of month.
<i>date</i>	A numeric constant, variable, expression, or the label of a STRING, CSTRING, or PSTRING variable declared with a date picture token. The <i>date</i> must be a standard date. A variable declared with a date picture token is automatically converted to a standard date intermediate value.

The **DAY** procedure computes the day of the month (1 to 31) for a given standard date.

Return Data Type: **LONG**

Example:

```
OutDay = DAY(TODAY())    !Get the day from today's date
DueDay = DAY(TODAY()+2) !Calculate the return day
```

See Also:

Standard Date

MONTH

YEAR

TODAY

DATE

DEFORMAT (return unformatted numbers from string)

DEFORMAT(*string* [,*picture*])

DEFORMAT Removes formatting characters from a numeric string.

string A string expression containing a numeric string.

picture A picture token, or the label of a CSTRING variable containing a picture token. If omitted, the picture for the *string* parameter is used. If the *string* parameter was not declared with a picture token, the return value will contain only characters that are valid for a numeric constant.

The **DEFORMAT** procedure removes formatting characters from a numeric string, returning only the numbers contained in the string. When used with a date or time *picture* (except those containing alphabetic characters), it returns a STRING containing the Clarion Standard Date or Time.

Return Data Type: STRING

Example:

```
!DEFORMAT('$1,234.56')           returns 1234.56
!DEFORMAT('309-53-9954')       returns 309539954
!DEFORMAT('40A1-7',@P##A1-#P)  returns 407
```

!Get phone number for modem to dial:

```
DialString = 'ATDT1' & DEFORMAT(Phone,@P(###)###-###P) & '<13,10>'
```

```
ClarionDate = DEFORMAT(dBaseDate,@D1) !Clarion Standard date from mm/dd/yy string
```

```
Data = '45,123' !Assign a formatted number to a string
Number = DEFORMAT(Data) !then remove non-numeric characters
```

See Also:

FORMAT

Standard Date

Standard Time

Picture Tokens

DELETE (delete a record)

DELETE(*entity*)

DELETE Removes a record from a FILE, VIEW, or QUEUE structure.

entity The label of a FILE, VIEW, or QUEUE structure.

The **DELETE** statement removes a record.

FILE Usage

DELETE(file) removes the last record successfully accessed by NEXT, PREVIOUS, GET, REGET, ADD, or PUT. The key entries for that record are also removed from the KEYS. DELETE does not clear the record buffer. Therefore, data values from the record just deleted still exist and are available for use until the record buffer is overwritten. If no record was previously accessed, or the record is held by another workstation, DELETE posts the "Record Not Available" error and no record is deleted.

VIEW Usage

DELETE(view) removes the last VIEW primary file record that was successfully accessed by a NEXT or PREVIOUS statement. The key entries for that record are also removed from the KEYS. DELETE does not remove records from any secondary JOIN files in the VIEW. If no record was previously accessed, or the record is held by another workstation, DELETE posts the "Record Not Available" error and no record is deleted. The specific disk action DELETE performs in the file is file driver dependent.

DELETE only deletes the primary file record in the VIEW because the VIEW structure performs both relational Project and Join operations at the same time. Therefore, it is possible to create a VIEW structure that, if all its component files were updated, would violate the Referential Integrity rules set for the database. The common solution to this problem in SQL-based database products is to delete only the Primary file record. Therefore, Clarion has adopted this same industry standard solution.

QUEUE Usage

DELETE(queue) removes the QUEUE entry at the position of the last successful GET or ADD and de-allocates its memory. If no previous GET or ADD was executed, the "Entry Not Found" error is posted. DELETE does not affect the current POINTER procedure return value, however, once the entry is deleted, the POINTER value for all subsequent entries in the QUEUE decrement by one (1).

If the QUEUE contains any reference variables or fields with the ANY data type, you must reference assign a NULL to each reference variable and ANY field in the queue structure before the DELETE statement. This will avoid memory leaks by freeing up the memory used by the ANY or reference variables before the DELETE statement.

```
Errors Posted:    05 Access Denied
                  08 Insufficient Memory
                  30 Entry Not Found
                  33 Record Not Available
```

Example:

```
Customer  FILE,DRIVER('Clarion'),PRE(Cus)  !Declare customer file layout
AcctKey   KEY(Cus:AcctNumber)
Record    RECORD
AcctNumber LONG
Name      STRING(20)
Addr      STRING(20)
City      STRING(20)
State     STRING(20)
Zip       STRING(20)
          END
          END

CustView  VIEW(Customer)                    !Declare VIEW structure
          PROJECT(Cus:AcctNumber,Cus:Name)
          END

NameQue   QUEUE,PRE(Que)
Name      STRING(20),NAME('FirstField')
Zip       DECIMAL(5,0),NAME('SecondField')
          END

CODE
DO BuildQue                                !Call routine to build the queue
LOOP X# = RECORDS(NameQue) TO 1 BY -1      !Loop backwards through queue
  GET(NameQue,X#)                           !getting each entry
  ASSERT(NOT ERRORCODE())
  IF NameQue.Name[1] = 'J'                   !Evaluate a condition
    DELETE(NameQue)                          !and delete only specific entries
    ASSERT(NOT ERRORCODE())
  END
END

OPEN(Customer)
Cus:AcctNumber = 12345                       !Initialize key field
SET(Cus:AcctKey,Cus:AcctKey)
```

```
OPEN(CustView)
NEXT(CustView)                !Get that record
  IF ERRORCODE()
    STOP(ERROR())
  END
DELETE(CustView)              !Delete the customer record
CLOSE(CustView)               !Close the VIEW

Cus:AcctNumber = 12345        !Initialize key field
GET(Customer,Cus:AcctKey)    !Get that record
IF ERRORCODE() THEN STOP(ERROR()).
DELETE(Customer)              !Delete the customer record
```

See Also:

HOLD, NEXT, PREVIOUS, PUT, GET, ADD, ANY, CLEAR, Reference Variables

DELETereg (remove a value or key from Windows registry)

DELETereg(LONG root, STRING keyname [,.STRING valuename])

DELETereg Deletes and named entry from the system registry.

root The root section of the registry from which to delete the value. Valid values for this are defined in equates.clw and are as follows:

```
REG_CLASSES_ROOT
REG_CURRENT_USER
REG_LOCAL_MACHINE
REG_USERS
REG_PERFORMANCE_DATA
REG_CURRENT_CONFIG
REG_DYN_DATA
```

keyname The name of the key to delete, or the key to delete from.

valuename The name of the value to delete. If omitted the whole key is deleted.

Example:

```
PROGRAM
MAP.

INCLUDE('EQUATES')
CurrentPath CSTRING(100)
ColorScheme CSTRING(100)

CODE
DELETereg(REG_LOCAL_MACHINE,'SOFTWARE\SoftVelocity\Clarion6','root')
!remove the root directory entry of the Clarion 6 install
DELETereg(REG_CURRENT_USER,'Control Panel\Current')
!since a specific value is not specified,
!the entire "Current" key will be removed
```

See Also: GETREG, PUTREG

DESTROY (remove a control or file)

```
DESTROY( [ first control ] [, last control ] )
        [ file ]
```

DESTROY	Removes window controls.
<i>first control</i>	Field number or field equate label of a control, or the first control in a range of controls. If omitted, defaults to zero (0).
<i>last control</i>	Field number or field equate label of the last control in a range of controls.
<i>file</i>	The label of the FILE to be destroyed, or variable containing the filename of the file to erase.

The **DESTROY** statement removes a control, or range of controls, from an APPLICATION or WINDOW structure. When removed, the control's resources are returned to the operating system.

DESTROYing a GROUP, OPTION, MENU, TAB, or SHEET control also destroys all controls contained within it.

DESTROY(File) is similar to CLOSE(File), but additionally kills any internal information stored by the runtime or driver libraries. For example, ODBC-based drivers can store the data source information associated with the table even after it has been closed to speed up its next reopening. Usually, this internal data uses the address of the FILE structure as a key. Therefore, if the FILE structure name has been changed to use with another table or process, incorrect information can be used.

The DESTROY(File) statement solves this problem.

Example:

```
Screen WINDOW, PRE(Scr)
    ENTRY(@N3), USE(Ctl:Code)
    ENTRY(@S30), USE(Ctl:Name)
    BUTTON('OK'), USE(?OkButton), KEY(EnterKey)
    BUTTON('Cancel'), USE(?CanxButton), KEY(EscKey)
END

CODE
OPEN(Screen)
DESTROY(?Ctl:Code)           !Remove a control
DESTROY(?Ctl:Code,?Ctl:Name) !Remove range of controls
DESTROY(2)                   !Remove the second control
```

```
!*****
!Example of use of DESTROY(File):
!without DESTROY(File):
  OPEN(File)
  ...
  !do something here
  ...
  CLOSE(File)
  File{PROP:Name} = NewFilename
  OPEN(File)                                !possible problem because the driver can use
                                           !data collected for the previous existence of
                                           !the table
  ...

!WITH DESTROY(File):

  OPEN(File)
  ...
  !do something
  ...
  DESTROY(File)
  File{PROP:Name} = NewFileName
  OPEN(File)                                !OK now
```

See Also:

CREATE

DIRECTORY (get file directory)

DIRECTORY(*queue*, *path*, *attributes*)

DIRECTORY Gets a file directory listing (just like the DIR command in DOS).

queue The label of the QUEUE structure that will receive the directory listing.

path A string constant, variable, or expression that specifies the path and filenames directory listing to get. This may include the wildcard characters (* and ?).

attributes An integer constant, variable, or expression that specifies the attributes of the files to place in the *queue*.

The **DIRECTORY** procedure returns a directory listing of all files in the *path* with the specified *attributes* into the specified *queue*.

The *queue* parameter must name a QUEUE with a structure that begins the same as the following structure contained in EQUATES.CLW:

```

ff_:queue  QUEUE,PRE(ff_),TYPE
name       STRING(13)
date       LONG
time       LONG
size       LONG
attrib     BYTE           !A bitmap, the same as the attributes EQUATES
END

```

or the following structure (for long filename support):

```

FILE:queue QUEUE,PRE(File),TYPE
name       STRING(FILE:MAXFILENAME) !FILE:MAXFILENAME is an EQUATE
shortname  STRING(13)
date       LONG
time       LONG
size       LONG
attrib     BYTE           !A bitmap, the same as the attributes EQUATES
END

```

Your QUEUE may contain more fields, but must begin with these fields. It will receive the returned information about each file in the *path* that has the *attributes* you specify. The date and time fields will contain standard Clarion date and time information (the conversion from the operating system's storage format to Clarion standard format is automatic).

The *attributes* parameter is a bitmap which specifies what filenames to place in the *queue*. The following equates are contained in EQUATES.CLW:

```
ff_:NORMAL      EQUATE(0)      !Always active
ff_:READONLY    EQUATE(1)      !Not for use as attributes parameter
ff_:HIDDEN      EQUATE(2)
ff_:SYSTEM      EQUATE(4)
ff_:DIRECTORY   EQUATE(10H)
ff_:ARCHIVE     EQUATE(20H)    ! NOT Win95 compatible
```

The *attributes* bitmap is an OR filter: if you add the equates, you get files with any of the attributes you specify. This means that, when you just set the *attributes* to ff_:NORMAL, you only get files (no sub-directories) without the hidden, system, or archive bits set. If you add ff_:DIRECTORY to ff_:NORMAL, you will get files AND sub-directories from the *path*. Since ff_:NORMAL is an equate for zero (0), you will always get files.

Short filenames used in applications

Applications should avoid the use of the short filenames returned by DIRECTORY. This includes *never* using the DIRECTORY(ff_:Queue...) syntax, which retrieves only short names. Instead, use the DIRECTORY(FILE:QUEUE...) syntax, and avoid using the *ShortName* field.

Short file names do not exist under all file systems. The short file name returned in the queue is sometimes the long file name, truncated to 13 bytes. Under Windows NT/2000/XP systems, the registry key **NtfsDisable8dot3NameCreation** allows turning off short file names. Files created after this key is enabled will not have a short file name generated.

Note:

Setting *NtfsDisable8dot3NameCreation=1* in the Windows system registry improves performance. This setting may become more popular and eventually can be the default. Developers using short file names in applications released to the general public may run into problems in the near future.

Tip

Sometimes you can run into a user running your applications on a Novell server *without* the long file name NLM loaded, so you can't rely on *all* file systems supporting long file names. Therefore, the safest strategy is to name your files with legal 8.3 short names, and as noted above, only use the long file name queue field returned by DIRECTORY.

Example:

```
DirectoryList PROCEDURE
```

```
AllFiles QUEUE(File:queue),PRE(FIL)      !Inherit exact declaration of File:queue
      END
```

```
LP      LONG
Recs    LONG
```

```
CODE
DIRECTORY(AllFiles,'*.*',ff_:DIRECTORY)  !Get all files and directories
Recs = RECORDS(AllFiles)
LOOP LP = Recs TO 1 BY -1
  GET(AllFiles,LP)
  IF BAND(FIL:Attrib,ff_:DIRECTORY) AND FIL:ShortName <> '..' AND FIL:ShortName <> '.'
    CYCLE                                !Let sub-directory entries stay
  ELSE
    DELETE(AllFiles)                     !Get rid of all other entries
  END
END
```

See Also:

SHORTPATH

LONGPATH

PATH

DISABLE (dim a control)

DISABLE([*first control*] [, *last control*])

DISABLE Dims controls on the window.

first control Field number or field equate label of a control, or the first control in a range of controls. If omitted, defaults to zero (0).

last control Field number or field equate label of the last control in a range of controls.

The **DISABLE** statement disables a control or a range of controls on an APPLICATION or WINDOW structure. When disabled, the control appears dimmed on screen.

Example:

```
Screen WINDOW,PRE(Scr)
    ENTRY(@N3),USE(Ctl:Code)
    ENTRY(@S30),USE(Ctl:Name)
    BUTTON('OK'),USE(?OkButton),KEY(EnterKey)
    BUTTON('Cancel'),USE(?CanxButton),KEY(EscKey)
END

CODE
OPEN(Screen)
DISABLE(?Ctl:Code)           !Disable a control
DISABLE(?Ctl:Code,?Ctl:Name)   !Disable range of controls
DISABLE(2)                   !Disable the second control
```

See Also:

ENABLE

HIDE

UNHIDE

DISPLAY (write USE variables to screen)

DISPLAY([*first control*] [,*last control*])

DISPLAY	Writes the contents of USE variables to their associated controls.
<i>first control</i>	Field number or field equate label of a control, or the first control in a range of controls.
<i>last control</i>	Field number or field equate label of the last control in a range of controls.

DISPLAY writes the contents of the USE variables to their associated controls on the active window. **DISPLAY** with no parameters writes the USE variables for all controls on the screen. Using *first control* alone, as the parameter of **DISPLAY**, writes a specific USE variable to the screen. Both *first control* and *last control* parameters are used to display the USE variables for an inclusive range of controls on the screen.

The current contents of the USE variables of all controls are automatically displayed on screen each time the **ACCEPT** loop cycles if the window has the **AUTO** attribute present. This eliminates the need to explicitly issue a **DISPLAY** statement to update the video display. Of course, if your application performs some operation that takes a long time and you want to indicate to the user that something is happening without cycling back to the top of the **ACCEPT** loop, you should **DISPLAY** some variable that you have updated.

Example:

```

DISPLAY                !Display all controls on the screen
DISPLAY(2)             !Display control number 2
DISPLAY(3,7)          !Display controls 3 through 7
DISPLAY(?MenuControl) !Display the menu control
DISPLAY(?TextBlock,?Ok) !Display range of controls

```

See Also:

Field Equate Labels

UPDATE

ERASE

CHANGE

AUTO

DRAGID (return matching drag-and-drop signature)

DRAGID([*thread*] [, *control*])

DRAGID	Returns matching host and target signatures on a successful drag-and-drop operation.
<i>thread</i>	The label of a numeric variable to receive the thread number of the host control. If the host control is in an external program, <i>thread</i> receives zero (0).
<i>control</i>	The label of a numeric variable to receive the field equate label of the host control.

The **DRAGID** procedure returns the matching host and target control signatures on a successful drag-and-drop operation. If the user aborted the operation, DRAGID returns an empty string ("), otherwise it returns the first signature that matched between the two controls.

Return Data Type: STRING

Example:

```

Que1 QUEUE
    STRING(30)
    END
Que2 QUEUE(Que1)          !Que2 declared same as Que1
    END
Que3 QUEUE(Que1)          !Que3 declared same as Que1
    END
WinOne WINDOW,AT(0,0,360,400)
    !Allows drags, but not drops:
    LIST,AT(120,0,20,20),USE(?List1),FROM(Que1),DRAGID('List1')
    !Allows drops from List1 or List3, but no drags:
    LIST,AT(120,120,20,20),USE(?List2),FROM(Que2),DROPID('List1','List3')
    !Allows drags, but not drops:
    LIST,AT(120,240,20,20),USE(?List3),FROM(Que3),DRAGID('List3')
    END
CODE
OPEN(WinOne)
ACCEPT
CASE EVENT()
  OF EVENT:Drop           !When drop event is successful
    CASE DRAGID()         !check for which host dropped it
      OF 'List1'
        Que2 = Que1       !get dropped info from Que1
      OF 'List3'
        Que2 = Que3       !get dropped info from Que3
    END
  ADD(Que2)               !add either one to the drop queue
END
END
END

```

See Also:

DROPID

SETDROPID

DROPID (return drag-and-drop string)

DROPID([*thread*] [, *control*])

DROPID	Returns matching host and target signatures on a successful drag-and-drop operation.
<i>thread</i>	The label of a numeric variable to receive the thread number of the target control. If the target control is in an external program, <i>thread</i> receives zero (0).
<i>control</i>	The label of a numeric variable to receive the field equate label of the target control.

The **DROPID** procedure returns the matching host and target control signatures on a successful drag-and-drop operation (just as DRAGID does), or the specific string set by the SETDROPID procedure. The DROPID procedure returns a comma-delimited list of filenames dragged from the Windows File Manager when '~FILE' is the DROPID attribute.

Return Data Type: STRING

Example:

```

DragDrop PROCEDURE
Que1 QUEUE
    STRING(90)
    END

Que2 QUEUE
    STRING(90)
    END

WinOne WINDOW('Test Drag Drop'),AT(10,10,240,320),SYSTEM,MDI
    !Drag but no drop:
    LIST,AT(12,0,200,80),USE(?List1),FROM(Que1),DRAGID('List1')
    !Allows drops from List1 or the Windows File Manager, but no drags:
    LIST,AT(12,120,200,80),USE(?List2),FROM(Que2),DROPID('List1','~FILE')
    END

CODE
OPEN(WinOne)
ACCEPT
CASE EVENT()
OF EVENT:Drag
    IF DRAGID()
        GET(Que1,CHOICE())
        SETDROPID(Que1)
    END
    !When a drag event is attempted
    !check for success
    !and setup info to pass
OF EVENT:Drop
    IF INSTRING(', ',DROPID(),1,1)
        Que2 = |
        SUB(DROPID(),1,INSTRING(', ',DROPID(),1,1)-1)
        ADD(Que2)
    ELSE
        Que2 = DROPID()
        ADD(Que2)
    END
    !When drop event is successful
    !Check for multiple files from File Manager
    !and only get first
    !and add it to the queue
    !get dropped info, from List1 or File Manager
    !and add it to the queue
END
END
END

```

DUPLICATE (check for duplicate key entries)

```
DUPLICATE( |key | )
          |file |
```

DUPLICATE Checks duplicate entries in unique keys.

key The label of a KEY declaration.

file The label of a FILE declaration.

The **DUPLICATE** procedure returns a non-zero value (true) if writing the current record to the data file would post the "Creates Duplicate Key" error. With a *key* parameter, only the specified KEY is checked. With a *file* parameter, all KEYS declared without a DUP attribute are checked. DUPLICATE is most useful to detect potential duplicate key errors before writing to disk.

The DUPLICATE procedure assumes that the contents of the RECORD structure data buffer are duplicated at the current record pointer location. Therefore, when using DUPLICATE prior to ADDing a record, the record pointer should be cleared with: GET(*file*,0).

Return Data Type: LONG

Example:

```
IF Action = 'ADD' THEN GET(Vendor,0).           !If adding, clear the file pointer
IF DUPLICATE(Vendor)                          !If this vendor already exists
  SCR:MESSAGE = 'Vendor Number already assigned' !display message
  SELECT(?)                                    !and stay on the field
END
```

See Also:

GET

ADD

DUP

ELLIPSE (draw an ellipse)

ELLIPSE(*x* , *y* , *width* , *height* [, *fill*] [, *attributelist*])

ELLIPSE	Draws an ellipse on the current window or report.
<i>x</i>	An integer expression that specifies the horizontal position of the starting point.
<i>y</i>	An integer expression that specifies the vertical position of the starting point.
<i>width</i>	An integer expression that specifies the width.
<i>height</i>	An integer expression that specifies the height.
<i>fill</i>	A LONG integer constant, constant EQUATE, or variable containing the red, green, and blue components that create the color in the three low-order bytes (bytes 0, 1, and 2) or an EQUATE for a standard Windows color value.
<i>attributelist</i>	A string constant, variable, or EQUATE containing an optional type of output document and its associated attributes. Only valid when the target is a REPORT. (See <i>EXTEND</i>)

The **ELLIPSE** procedure places an ellipse on the current window or report. The ellipse is drawn inside a "bounding box" defined by the *x*, *y*, *width*, and *height* parameters. The *x* and *y* parameters specify the starting point, and the *width* and *height* parameters specify the horizontal and vertical size of the "bounding box."

The border color is the current pen color set by **SETPENCOLOR**; the default color is the Windows system color for window text. The border width is the current width set by **SETPENWIDTH**; the default width is one pixel. The border style is the current pen style set by **SETPENSTYLE**; the default style is a solid line.

Example:

```
MDIChild WINDOW('Child One'), AT(0,0,320,200), MDI, MAX, HVSCROLL
    !window controls
END
CODE
OPEN(MDIChild)
ELLIPSE(100,50,100,50,00FF0000h)    !Red ellipse
```

See Also: **Current Target**, **SETPENCOLOR**, **SETPENWIDTH**, **SETPENSTYLE**

EMPTY (empty a data file)

EMPTY(*file*)

EMPTY Deletes all records from a FILE.

file The label of a FILE.

EMPTY deletes all records from the specified *file*. **EMPTY** requires exclusive access to the file. Therefore, the file must be opened with *access mode* set to 12h (Read/Write Deny All) or 22h (Read/Write Deny Write).

Errors Posted: 63 Exclusive Access Required
 37 File Not Open

Example:

```
OPEN(Master,18)     !Open the master file
EMPTY(Master)      !and start a new one
```

See Also:

OPEN

SHARE

ENABLE (re-activate dimmed control)

ENABLE([*first control*] [, *last control*])

ENABLE Reactivates disabled controls.

first control Field number or field equate label of a control, or the first control in a range of controls. If omitted, defaults to zero (0).

last control Field number or field equate label of the last control in a range of controls.

The **ENABLE** statement reactivates a control, or range of controls, that were dimmed by the **DISABLE** statement, or were declared with the **DISABLE** attribute. Once reactivated, the control is again available to the operator for selection.

Example:

```
CODE
OPEN(Screen)
DISABLE(?Control2)            !Control2 is deactivated
IF Ctl:Password = 'Supervisor'
  ENABLE(?Control2)            !Re-activate Control2
END
```

See Also:

DISABLE

HIDE

UNHIDE

ENDPAGE (force page overflow)

ENDPAGE(*report* [, *printfooters*])

ENDPAGE	Forces page overflow.
<i>report</i>	The label of a REPORT structure.
<i>printfooters</i>	An integer constant or variable. If omitted or zero (0), this prints group footers by forcing a group BREAK (usually used to terminate a report for preview). If one (1), group BREAKs are left open and no group footers print (usually used during a continuing report).

The **ENDPAGE** statement initiates page overflow and flushes the print engine's print structure buffer. If the REPORT has the PREVIEW attribute, this has the effect of ensuring that the entire report is available to view.

Example:

SomeReport PROCEDURE

```
WMFQue QUEUE                                !Queue to contain .WMF filenames
    STRING(64)
END
```

```
NextEntry  BYTE(1)                          !Queue entry counter variable
```

```
Report     REPORT,PREVIEW(WMFQue)           !Report with PREVIEW attribute
DetailOne  DETAIL
    !Report controls
    END
END
```

```
ViewReport WINDOW('View Report'),AT(0,0,320,200),MDI,MAX,HVSCROLL
    IMAGE('',AT(0,0,320,180),USE(?ImageField)
    BUTTON('View Next Page'),AT(0,180,60,20),USE(?NextPage),DEFAULT
    BUTTON('Print Report'),AT(80,180,60,20),USE(?PrintReport)
    BUTTON('Exit Without Printing'),AT(160,180,60,20),USE(?ExitReport)
END
```

```

CODE
OPEN(Report)
SET(SomeFile)                               !Code to generate the report
LOOP
  NEXT(SomeFile)
  PRINT(DetailOne)
END
ENDPAGE(Report)                             !Flush the buffer
OPEN(ViewReport)                            !Open report preview window
GET(WMFQue,NextEntry)                       !Get first queue entry
?ImageField{PROP:text} = WMFQue            !Load first report page
ACCEPT
CASE ACCEPTED()
OF ?NextPage
  NextEntry += 1                            !Increment entry counter
  IF NextEntry > RECORDS(WMFQue) THEN CYCLE. !Check for end of report
  GET(WMFQue,NextEntry)                     !Get next queue entry
  ?ImageField{PROP:text} = WMFQue           !Load next report page
  DISPLAY                                   !and display it
OF ?PrintReport
  Report{PROP:flushpreview} = ON            !Flush files to printer
  BREAK                                     !and exit procedure
OF ?ExitReport
  BREAK                                     !Exit procedure
. .
CLOSE(ViewReport)                           !Close window
FREE(WMFQue)                                 !Free the queue memory
CLOSE(Report)                               !Close report (deleting all .WMF files)
RETURN                                       ! and return to caller

```

See Also:

Page Overflow

PREVIEW

ERASE (clear screen control and USE variables)

ERASE([*first control*] [,*last control*])

ERASE	Blanks controls and clears their USE variables.
<i>first control</i>	Field number or field equate label of a control, or the first control in a range of controls. If omitted, defaults to zero (0).
<i>last control</i>	Field number or field equate label of the last control in a range of controls.

The **ERASE** statement erases the data from controls in the window and clears their corresponding USE variables. ERASE with no parameters erases all controls in the window. Using *first control* alone, as the parameter of ERASE, clears a specific USE variable and its associated control. Both *first control* and *last control* parameters are used to clear the USE variables and associated controls for an inclusive range of controls in the window.

Example:

```
ERASE(?)           !Erase the currently selected control
ERASE              !Erase all controls on the screen
ERASE(3,7)        !Erase controls 3 through 7
ERASE(?Name,?Zip) !Erase controls from name through zip
ERASE(?City,?City+2) !Erase City and 2 controls following City
```

See Also:

Field Equate Labels

CHANGE

ERROR (return error message)

ERROR()

The **ERROR** procedure returns a string containing a description of any error that was posted. If no error was posted, ERROR returns an empty string. You should interrogate ERROR immediately after the statement which you suspect may post an error because processing any other statement which *could* post an error will clear the internal errorcode.

Return Data Type: **STRING**

Example:

```
PUT(NameQueue)           !Write the record
IF ERROR() = 'Queue Entry Not Found' !If not found
  ADD(NameQueue)         !add new entry
  IF ERRORCODE()        !Check for unexpected error
    STOP(ERROR())
  END
END
```

See Also:

ERRORCODE

ERRORFILE

FILEERROR

FILEERRORCODE

POPERRORS

PUSHERRORS

Trappable Run Time Errors

ERRORCODE (return error code number)

ERRORCODE()

The **ERRORCODE** procedure returns the code number for any error that was posted. If no error was posted, **ERRORCODE** returns zero. You should interrogate **ERRORCODE** immediately after the statement that you suspect may post an error because processing any other statement that could post an error will clear the internal error code.

Return Data Type: LONG

Example:

```
ADD(Location)           !Add new entry
IF ERRORCODE() = 8      !If not enough memory
  MESSAGE('Out of Memory') !display message
END
```

See Also:

ERROR

ERRORFILE

FILEERROR

FILEERRORCODE

POPERRORS

PUSHERRORS

Trappable Run Time Errors

ERRORFILE (return error filename)

ERRORFILE()

The **ERRORFILE** procedure returns the name of the file for which an error was posted. If the file is open, the full DOS file specification is returned. If the file is not open, the contents of the FILE statement's NAME attribute is returned. If the file is not open and the file has no NAME attribute, the label of the FILE statement is returned. If no error was posted, or the posted error did not involve a file, ERRORFILE returns an empty string.

Return Data Type: **STRING**

Example:

```
ADD(Location)                                !Add new entry
IF ERRORCODE()
  MESSAGE('Error with ' & ERRORFILE()) !Display error filename
END
```

See Also:

ERRORCODE

ERROR

FILEERRORCODE

FILEERROR

POPERRORS

PUSHERRORS

Trappable Run Time Errors

EVENT (return event number)

EVENT()

The **EVENT** procedure returns a number indicating what caused ACCEPT to alert the program that something has happened that it may need to handle. There are EQUATEs listed in EQUATES.CLW for all the events the program may need to handle.

There are two types of events generated by ACCEPT: field-specific and field-independent events. Field-specific events affect a single control, while field-independent events affect the window or program. The type of event can be determined by the values returned by the ACCEPTED, SELECTED, and FIELD procedures. If you need to know which field has input focus on a field-independent event, use the FOCUS procedure.

For field-specific events:

The FIELD procedure returns the field number of the control on which the event occurred.

The ACCEPTED procedure returns the field number if the event is EVENT:Accepted. The SELECTED procedure returns the field number if the event is EVENT:Selected.

For field-independent events:

The FIELD, ACCEPTED, and SELECTED procedures all return zero (0).

Return Data Type: SIGNED

Example:

```
ACCEPT
CASE EVENT()
OF EVENT:Selected
CASE SELECTED()
OF ?Control1
!Pre-edit code here
OF ?Control2
!Pre-edit code here
END
OF EVENT:Accepted
CASE ACCEPTED()
OF ?Control1
!Post-edit code here
OF ?Control2
!Post-edit code here
END
OF EVENT:Suspend
!Save some stuff
OF EVENT:Resume
!Restore the stuff
END
END
```

See Also: ACCEPT, FIELD, FOCUS, ACCEPTED, SELECTED

EXISTS (return file existence)

EXISTS(*file*)

EXISTS Returns TRUE if the *file* is available on disk.

file An expression containing the DOS filename.

The **EXISTS** procedure returns true (1) if the *file* is available on disk. If the *file* is not available, EXISTS returns false (0).

Return Data Type: LONG

Example:

```
IF EXISTS(SomeFile)
  OPEN(SomeFile)
ELSE
  CREATE(SomeFile)
END
```

FIELD (return control with focus)

FIELD()

The **FIELD** procedure returns the field number of the control which has focus at the time of any field-specific event. This includes both the EVENT:Selected and EVENT:Accepted events. FIELD returns zero (0) for field-independent events.

Positive field numbers are assigned by the compiler to all WINDOW controls, in the order their declarations occur in the WINDOW structure. Negative field numbers are assigned to all APPLICATION controls. In executable code statements, field numbers are usually represented by field equate labels--the label of the USE variable preceded by a question mark (?FieldName).

Return Data Type: SIGNED

Example:

```
Screen WINDOW
    ENTRY(@N4),USE(Control1)
    ENTRY(@N4),USE(Control2)
    ENTRY(@N4),USE(Control3)
    ENTRY(@N4),USE(Control4)
END

CODE
ACCEPT
    IF NOT ACCEPTED()
        CYCLE
    END
CASE FIELD()
    OF ?Control1
        !Control edit control
        !Field number 1
        IF Control1 = 0
            !if no entry
            BEEP
            !sound alarm
            SELECT(?)
            !stay on control
        END
    OF ?Control2
        !Field number 2
        IF Control2 > 4
            !if status is more than 4
            Scr:Message = 'Control must be less than 4'
            ERASE(?)
            !clear control
            SELECT(?)
            !edit the control again
        ELSE
            !value is valid
            CLEAR(Scr:Message)
            !clear message
        END
    OF ?Control4
        !Field number 4
        BREAK
        !exit processing loop
    END
END
!end case, end loop
```

See Also: ACCEPT, ACCEPTED, SELECTED, FOCUS, EVENT

FILEDIALOG (return chosen file)

FILEDIALOG([*title*] ,*file* [,*extensions*] [,*flag*])

FILEDIALOG	Displays Windows standard file choice dialogs to allow the user to choose a file.
<i>title</i>	A string constant or variable containing the title to place on the dialog. If omitted, Windows supplies a default.
<i>file</i>	The label of the string variable to receive the selected filename(s).
<i>extensions</i>	A string constant or variable containing the available file extension selections for the "List Files of Type" drop list. If omitted, the default is all files (*.*) .
<i>flag</i>	An integer constant or variable containing a bitmap to indicate the type of file action to perform.

The **FILEDIALOG** procedure displays Windows standard file choice dialogs and returns the file chosen by the user in the *file* parameter. Any existing value in the *file* parameter sets the default file choice presented to the user in the file choice dialog.

FILEDIALOG displays either the standard *Open...* dialog or the standard *Save...* dialog. By default, on the *Open...* dialog, the user is warned if the file they choose does not exist and the file is not opened. On the *Save...* dialog, the user is warned if the file does exist and the file is not saved.

The *extensions* parameter string must contain a description followed by the file mask. All elements in the string must be delimited by the vertical bar (|) character. For example, the *extensions* string:

```
'All Files | *.* | Clarion Source | *.CLW;*.INC;*.EQU;*.INT | Clarion Templates|*.TPL;*.TPW'
```

defines three selections for the "List Files of Type" drop list. The first extension listed in the *extensions* string is the default. Multiple extensions are separated by a semicolon (;) character.

The *flag* parameter is a bitmap that indicates the type of file action to perform (see EQUATES.CLW for symbolic constants). For bit number:

- 0 If zero (0000b), the *Open...* dialog displays.
- 1 If one (0001b), the *Save...* dialog displays.
- 2 If two (0010b), saves and restores the current directory path.
- 4 If four (0100b), doesn't report errors if the file does exist on *Save...* or does not exist on *Open...*
- 8 If eight (1000b), returns multiple selections when the user selects multiple files. When using long filename dialog, it returns a vertical bar (|) delimited string of filenames (with the full path on the string). The string is space-delimited when using short filename dialog. Not valid when used with File:Directory.
- 16 If sixteen (10000b), uses long filename dialog in 32-bit programs.

32 If thirty-two (10000b), displays a directory select dialog for selecting a directory path.

The following is a comprehensive sample of these (see EQUATES.CLW for the complete list):

FILE:Save	EQUATE(1)
FILE:KeepDir	EQUATE(2)
FILE:NoError	EQUATE(4)
FILE:Multi	EQUATE(8)
FILE:LongName	EQUATE(10H)
FILE:Directory	EQUATE(20H)

FILEDIALOG returns zero (0) if the user pressed the Cancel button, or one (1) if the user pressed the Ok button on the file choice dialog. If the user changes directories using the file dialog, your application's current directory also changes (unless you set FILE:KeepDir). This is a feature of the Windows operating system. If you do not want users to change your application's current directory but do want them to be able look in other directories, either save the current directory with the PATH() procedure before calling FILEDIALOG then restore it with the SETPATH() statement, or set FILE:KeepDir.

Return Data Type: **BOOL**

Example:

```
ViewTextFile PROCEDURE
ViewQue QUEUE                !LIST control display queue
    STRING(255)
END

FileName  STRING(64),STATIC  !Filename variable

ViewFile  FILE,DRIVER('ASCII'),NAME(FileName),PRE(Vew)
Record    RECORD
    STRING(255)
    END
    END

MDIChild1 WINDOW('View Text File'),AT(0,0,320,200),MDI,SYSTEM,HVSCROLL
    LIST,AT(0,0,320,200),USE(?L1),FROM(ViewQue),HVSCROLL
    END

CODE
IF NOT FILEDIALOG|
('Choose File to View',FileName,'Text|*.TXT|Source|*.CLW',FILE:LongName)
RETURN                !Return if no file chosen
END
OPEN(ViewFile)        !Open the file
IF ERRORCODE() THEN RETURN END!aborting on any error
```

```

SET(ViewFile)                !Start at top of file
LOOP
  NEXT(ViewFile)              !Reading each line of text
  IF ERRORCODE() THEN BREAK END !Break loop at end of file
  ViewQue = Vew:Record        !Assign text to queue
  ADD(ViewQue)                !and add a queue entry
END
CLOSE(ViewFile)              !Close the file
OPEN(MDICHild1)              !and open the window
ACCEPT                        !Allow the user to read the text and
END                            !break out of ACCEPT loop only from
                                !system menu close option

FREE(ViewQue)                !Free the queue memory
RETURN                        !and return to caller

!*****

!This example shows using FILEDIALOG for multi-file selection:

SelectFiles  PROCEDURE(SelectFileQueue DFQ)

Found        CSTRING(10000),AUTO
Path         CSTRING(File:MaxFilePath),AUTO
Separator    STRING(1),AUTO
Pos          UNSIGNED,AUTO
NameStart    UNSIGNED,AUTO

CODE
Found=SELF.DefaultFile
IF FILEDIALOG('Pick 1 or more files',Found,|
'All Files | *.* | Clarion Source | *.CLW;*.INC;*.EQU;*.INT|Clarion Templates|*.TPL;*.TPW',|
FILE:KeepDir+FILE:Multi+FILE:LongName)
  Separator='|'
  Pos=INSTRING(Separator,Found,1,1)
  IF Pos                                !Multi-Selected files
    ASSERT(Pos > 1)
    Path = CHOOSE(Found[Pos-1] <> '\', Found[1 : Pos-1]&'\', Found[1 : Pos-1])
    LOOP
      NameStart = Pos+1
      Pos = INSTRING(Separator,Found,1,NameStart)
      IF ~Pos THEN Pos=LEN(Found)+1.
      DFQ.Name = Path&Found[NameStart : Pos-1]
      ADD(DFQ)
    WHILE Pos<=LEN(Found)
  ELSE
    DFQ.Name=Found                                !Single file only selected
    ADD(DFQ)
  END
END
END

```

See Also: SETPATH, SHORTPATH, LONGPATH, DIRECTORY

FILEDIALOGA (extended file dialog)

FILEDIALOGA([*title*] ,*file* [,*extensions*] [,*flag*] [,*index*])

FILEDIALOGA Displays Windows standard file choice dialogs to allow the user to choose a file.

<i>title</i>	A string constant or variable containing the title to place on the dialog. If omitted, Windows supplies a default.
<i>file</i>	The label of the string variable to receive the selected filename(s).
<i>extensions</i>	A string constant or variable containing the available file extension selections for the "List Files of Type" drop list. If omitted, the default is all files (*.*) .
<i>flag</i>	An integer constant or variable containing a bitmap to indicate the type of file action to perform.
<i>index</i>	A signed integer variable used to select a different default extension by specifying the index number.

The **FILEDIALOGA** procedure displays Windows standard file choice dialogs and returns the file chosen by the user in the *file* parameter. Any existing value in the *file* parameter sets the default file choice presented to the user in the file choice dialog.

FILEDIALOGA displays either the standard *Open...* dialog or the standard *Save...* dialog. By default, on the *Open...* dialog, the user is warned if the file they choose does not exist and the file is not opened. On the *Save...* dialog, the user is warned if the file does exist and the file is not saved.

The *extensions* parameter string must contain a description followed by the file mask. All elements in the string must be delimited by the vertical bar (|) character. For example, the *extensions* string:

```
'All Files | *.* | Clarion Source | *.CLW;*.INC;*.EQU;*.INT | Clarion Templates|*.TPL;*.TPW'
```

defines three selections for the "List Files of Type" drop list. The first extension listed in the *extensions* string is the default. Multiple extensions are separated by a semicolon (;) character.

The *flag* parameter is a bitmap that indicates the type of file action to perform (see EQUATES.CLW for symbolic constants). For bit number:

- 0 If zero (0000b), the *Open...* dialog displays.
- 1 If one (0001b), the *Save...* dialog displays.
- 2 If two (0010b), saves and restores the current directory path.
- 4 If four (0100b), doesn't report errors if the file does exist on *Save...* or does not exist on *Open...*
- 8 If eight (1000b), returns multiple selections when the user selects multiple files. When using long filename dialog, it returns a vertical bar (|) delimited string of filenames (with the full path on the first). The string is space-delimited when using short filename dialog. Not valid when used with File:Save or File:Directory.
- 16 If sixteen (10000b), uses long filename dialog in 32-bit programs.
- 32 If thirty-two (100000b), displays a directory select dialog for selecting a directory path.

The following is a comprehensive sample of these (see EQUATES.CLW for the complete list):

FILE:Save	EQUATE(1)
FILE:KeepDir	EQUATE(2)
FILE:NoError	EQUATE(4)
FILE:Multi	EQUATE(8)
FILE:LongName	EQUATE(10H)
FILE:Directory	EQUATE(20H)

Finally, the *index* parameter is a SIGNED integer used to specify a different default extension other than the first one in the *extensions* list. For example:

```
'All Files | *.* | Clarion Source | *.CLW;*.INC;*.EQU;*.INT | Clarion Templates|*.TPL;*.TPW'
```

The extensions string consists of pairs: <description>|<mask>, and the index parameter references such pairs. Therefore, in the example above, an *index* value of 2 will display the '*Clarion Source*' description and associated extensions; an index value of 3 will display '*Clarion Templates*', etc.

If **FILEDIALOGA** returns a non-zero value (i.e., the dialog is completed with Open or Save button), the *index* parameter is set to the index of the extension pair that was used for file selection.

FILEDIALOGA returns zero (0) if the user pressed the Cancel button, or one (1) if the user pressed the Ok button on the file choice dialog. If the user changes directories using the file dialog, your application's current directory also changes (unless you set FILE:KeepDir). This is a feature of the Windows operating system. If you do not want users to change your application's current directory but do want them to be able look in other directories, either save the current directory with the PATH() procedure before calling **FILEDIALOGA** then restore it with the SETPATH() statement, or set FILE:KeepDir.

Return Data Type: BOOL

Example:

```

ViewTextFile PROCEDURE
ViewQue QUEUE                !LIST control display queue
    STRING(255)
    END

FileName  STRING(64),STATIC  !Filename variable

ViewFile  FILE,DRIVER('ASCII'),NAME(FileName),PRE(Vew)
Record    RECORD
    STRING(255)
    END
    END

MDIChild1 WINDOW('View Text File'),AT(0,0,320,200),MDI,SYSTEM,HVSCROLL
    LIST,AT(0,0,320,200),USE(?L1),FROM(ViewQue),HVSCROLL
    END

CODE
IF NOT FILEDIALOGA|
('Choose File to View',FileName,'Text|*.TXT|Source|*.CLW',FILE:LongName,2)
    RETURN                !Return if no file chosen
    END
OPEN(ViewFile)           !Open the file
IF ERRORCODE() THEN RETURN END!aborting on any error
SET(ViewFile)           !Start at top of file
LOOP
    NEXT(ViewFile)       !Reading each line of text
    IF ERRORCODE() THEN BREAK END !Break loop at end of file
    ViewQue = Vew:Record !Assign text to queue
    ADD(ViewQue)         !and add a queue entry
    END
CLOSE(ViewFile)         !Close the file
OPEN(MDIChild1)        !and open the window
ACCEPT                 !Allow the user to read the text and
    END                !break out of ACCEPT loop only from
                        !system menu close option

FREE(ViewQue)          !Free the queue memory
RETURN                !and return to caller

```

```
!*****
```

```

PROGRAM
MAP
END

FN          STRING(260)
ExtPickInOut SIGNED,AUTO
ExtFound    SIGNED,AUTO

CODE
  ExtPickInOut = 2 !default to CSV
  IF ~FILEDIALOGA('Pick file to save',fn,'Text|*.TXT|Comma Delimited |
(*.CSV)|*.CSV|Tab Delimited (*.TSV)|*.TSV', FILE:Save+ FILE:LongName, |
  ExtPickInOut)
    RETURN
  END
  ExtFound = INLIST(UPPER(SUB(CLIP(FN),-4,4)),'.TXT','.CSV','.TSV')
  IF ~ExtFound !No extension so add one based on selected file type
    ExtFound=ExtPick
    fn=clip(fn) & CHOOSE(ExtPickInOut, '.TXT', '.CSV', '.TSV', '')
  END

!This code assumes a 3 byte extension. There are a other possible ways to test
!for an extension.

```

See Also:

SETPATH

SHORTPATH

LONGPATH

DIRECTORY

FILEERROR (return file driver error message)

FILEERROR()

The **FILEERROR** procedure returns a string containing the "native" error message from the file system (file driver) being used to access a data file. Valid only when **ERRORCODE()** = 90.

Return Data Type: **STRING**

Example:

```
PUT(NameFile)           !Write the record
IF ERRORCODE() = 90     !Back-end-specific error occurred
  MESSAGE(FILEERROR())
RETURN
END
```

See Also:

ERRORCODE

ERROR

ERRORFILE

FILEERRORCODE

Trappable Run Time Errors

FILEERRORCODE (return file driver error code number)

FILEERRORCODE()

The **FILEERRORCODE** procedure returns a string containing the code number for the "native" error message from the file system (file driver) being used to access a data file. Valid only when **ERRORCODE() = 90**.

Return Data Type: **STRING**

Example:

```
PUT(NameFile)           !Write the record
IF ERRORCODE() = 90     !Back-end-specific error occurred
  MESSAGE(FILEERRORCODE( ))
RETURN
END
```

See Also:

FILEERROR

ERRORFILE

ERRORCODE

ERROR

Trappable Run Time Errors

FIRSTFIELD (return first window control)

FIRSTFIELD()

The **FIRSTFIELD** procedure returns the lowest field number in the currently active window (or REPORT) as specified by SETTARGET. This does not include any controls in a TOOLBAR or MENUBAR or any controls created after the window is opened.

Return Data Type: SIGNED

Example:

```
DISABLE(FIRSTFIELD(),LASTFIELD()) !Dim all control fields
```

See Also:

LASTFIELD

FLUSH (flush buffers)

FLUSH(*file*)

FLUSH Terminates a STREAM operation, flushing the operating system buffers to disk, or flushes the BUFFER statement's buffers.

file The label of a FILE or VIEW.

The **FLUSH** statement terminates a STREAM operation. It flushes the operating system buffers, which updates the directory entry for that *file*. FLUSH will also flush the file driver's buffers allocated by the BUFFER statement. If both STREAM and BUFFER are active, all buffers are flushed.

Support for this statement is dependent upon the file system and its specific action is described in the file driver documentation (if different from that described here).

Example:

```

STREAM(History)           !Use DOS buffering
SET(Current)              !Set to top of current file
LOOP
  NEXT(Current)
  IF ERRORCODE() THEN BREAK END
  His:Record = Cur:Record
  ADD(History)
END
FLUSH(History)            !End streaming, flush buffers
OPEN(MyView)
BUFFER(MyView,10,5,2,300) !10 records per page, 5 pages behind and 2 read-ahead,
                          !with a 5 minute timeout
                          !Process records
FLUSH(MyView)             !Flush buffers

```

See Also:

STREAM

BUFFER

FOCUS (return control with focus)

FOCUS()

The **FOCUS** procedure returns the field number of the control which has input focus at any time any event occurs. Positive field numbers are assigned by the compiler to all WINDOW controls, in the order their declarations occur in the WINDOW structure. Negative field numbers are assigned to all APPLICATION controls. In executable code statements, field numbers are usually represented by field equate labels--the label of the USE variable preceded by a question mark (?FieldName).

Return Data Type: SIGNED

Example:

```
Screen WINDOW
    ENTRY(@N4),USE(Control1)
    ENTRY(@N4),USE(Control2)
    ENTRY(@N4),USE(Control3)
END

CODE
ACCEPT
CASE EVENT()
  OF EVENT:LoseFocus
  OROF EVENT:CloseWindow
    CASE FOCUS()      !Control edit control
      OF ?Control1   !Field number 1
        UPDATE(?Control1)
      OF ?Control2   !Field number 2
        UPDATE(?Control2)
      OF ?Control3   !Field number 3
        UPDATE(?Control3)
    END
  END
END
```

See Also:

ACCEPTED

SELECTED

FIELD

EVENT

FONTDIALOG (return chosen font)

FONTDIALOG(*[title]* ,*typeface* [*,size*] [*,color*] [*,style*] [*,added*])

FONTDIALOG	Displays the standard Windows font choice dialog box to allow the user to choose a font.
<i>title</i>	A string constant or variable containing the title to place on the font choice dialog. If omitted, a default <i>title</i> is supplied by Windows.
<i>typeface</i>	A string variable to receive the name of the chosen font.
<i>size</i>	An integer variable to receive the size (in points) of the chosen font.
<i>color</i>	A LONG integer variable to receive the red, green, and blue values for the color of the chosen font in the low-order three bytes.
<i>style</i>	An integer variable to receive the strike weight and style of the chosen font.
<i>added</i>	An integer constant or variable that specifies adding screen or printer fonts, or both, to the list of available fonts. Zero (0) adds screen fonts, one (1) adds printer fonts, and two (2) adds both. If omitted, only Windows registered fonts are listed.

The **FONTDIALOG** procedure displays the Windows standard font choice dialog box to allow the user to choose a font. When called, any values in the parameters set the default font values presented to the user in the font choice dialog. They also receive the user's choice when the user presses the Ok button on the dialog. FONTDIALOG returns zero (0) if the user pressed the Cancel button, or one (1) if the user pressed the Ok button.

Return Data Type: SHORT

Example:

```
MDIChild1  WINDOW('View Text File'),AT(0,0,320,200),MDI,SYSTEM,HVSCROLL
           !window controls
           END
Typeface   STRING(31)
FontSize   LONG
FontColor  LONG
FontStyle  LONG
CODE
OPEN(MDIChild1)                                !open the window
IF FONTDIALOG('Choose Display Font',Typeface,FontSize,FontColor,FontStyle,0)
  SETFONT(0,Typeface,FontSize,FontColor,FontStyle) !Set window font
ELSE
  SETFONT(0,'Arial',12)                          !Set default font
END
ACCEPT
!Window handling code
END
```


FORMAT (return formatted numbers into a picture)

FORMAT(*value*,*picture*)

FORMAT Returns a formatted numeric string.

value A numeric expression for the *value* to be formatted.

picture A picture token or the label of a CSTRING variable containing a picture token.

The **FORMAT** procedure returns a numeric string formatted according to the *picture* parameter.

Return Data Type: STRING

Example:

!Format the Social Security number:

```
Rpt:SocSecNbr = FORMAT(Emp:SSN,@P###-##-####P)
```

!Change phone format from dashes to parenthesis:

```
Phone = FORMAT(DEFORMAT(Phone,@P###-###-####P),@P(###)###-####P)
```

!Format a date as a string:

```
DateString = FORMAT(DateLong,@D1)
```

See Also:

DEFORMAT

Picture Tokens

FORWARDKEY (pass keystrokes to control)

FORWARDKEY(*targetcontrol*)

FORWARDKEY Passes a keystroke to another control.

targetcontrol A field number or field equate label for the control to receive the passed keystroke.

FORWARDKEY passes a keystroke to another control. This imitates the action of the keystroke as if the target control was the control in focus when the keystroke occurred.

Example:

```
FORWARDKEY(?LIST)           !pass the keystroke to the list box
```

FREE (delete all entries)

FREE(*queue*)

FREE Deletes all entries from a QUEUE.

queue The label of a QUEUE structure, or the label of a passed QUEUE parameter.

FREE deletes all entries from a QUEUE and de-allocates the memory they occupied. It also de-allocates the memory used by the QUEUE's "overhead." **FREE** does not clear the QUEUE's data buffer.

If the QUEUE contains any reference variables or fields with the ANY data type, you must first **CLEAR** each QUEUE entry before **FREE**ing the QUEUE. This will avoid memory leaks by freeing up the memory used by the ANY variables before the **FREE** statement removes the pointer to the allocated memory.

Errors Posted: 08 Insufficient Memory

Example:

```
FREE(Location)      !Free the location queue
FREE(NameQue)      !Free the name queue
```

See Also:

ANY

CLEAR

Reference Variables

FREESTATE (free resources)

FREESTATE(*file, savedstate*)

FREESTATE Releases all resources allocated by GETSTATE

file The label of a FILE declaration.

savedstate A LONG numeric variable which represents the value returned by the GETSTATE procedure.

FREESTATE releases all resources allocated by GETSTATE.

Example:

```
MyFunction PROCEDURE(FILE MyFile)
```

```
CurState LONG
```

```
CODE
```

```
! File will be in the same state on exit of this procedure as it was on entry
CurState = GETSTATE(MyFile)      !save current state of file
SET(MyFile)                       !ready to access file
NEXT(MyFile)                      !read a record
CLEAR(MyFile)                     !clear record buffer
RESTORESTATE(MyFile, CurState)    !restore file to initial state
FREESTATE(MyFile, CurState)       !release resources
```

See Also:

GETSTATE

RESTORESTATE

GET (read a record or entry)

```

      | file , key          |
GET(  | file , filepointer [, length ] | )
      | key , keypointer   |
      | queue , pointer   |
      | queue , [+]key,...[-]key |
      | queue , name      |
      | queue , function |

```

GET	Retrieves a specific record from a FILE or entry from a QUEUE.
<i>file</i>	The label of a FILE declaration.
<i>key</i>	The label of a KEY or INDEX declaration.
<i>filepointer</i>	A numeric constant, variable, or expression for the value returned by the POINTER(<i>file</i>) procedure.
<i>length</i>	An integer constant, variable, or expression which contains the number of bytes to read from the <i>file</i> . The <i>length</i> must be greater than zero and not greater than the RECORD length. If omitted or out of range, <i>length</i> defaults to the length of the RECORD structure.
<i>keypointer</i>	A numeric constant, variable, or expression for the value returned by the POINTER(<i>key</i>) procedure.
<i>queue</i>	The label of a QUEUE structure.
<i>pointer</i>	A numeric constant, variable, or numeric expression. The <i>pointer</i> must be in the range from 1 to the number of entries in the memory queue.
+ -	The leading plus or minus sign specifies the <i>key</i> is sorted in ascending or descending sequence.
<i>key</i>	The label of a field declared within the QUEUE structure. If the QUEUE has a PRE attribute, the <i>key</i> must include the prefix.
<i>name</i>	A string constant, variable, or expression containing the NAME attribute of QUEUE fields, separated by commas, and optional leading + or - signs for each attribute. This parameter is case sensitive.
<i>function</i>	The label of the function containing two parameters of a *GROUP or named GROUP passed by address, and having a SIGNED return value. Both parameters must use the same parameter type, and cannot be omitted. The RAW, C and PASCAL attributes are not permitted in the prototype declaration. See Additional Queue Considerations.

The **GET** statement locates a specific record in a FILE or specific entry in a QUEUE and retrieves it.

FILE Usage

The GET statement locates a specific record in the data file and reads it into the RECORD structure data buffer. Direct access to the record is achieved by relative record position within the file, or by matching key values. If the GET is unsuccessful, the previous content of the RECORD buffer is not affected.

GET(*file,key*)

Gets the first record from the file (as listed in the *key*) which contains values matching the values in the component fields of the *key*.

GET(*file,filepointer* [,*length*])

Gets a record from the file based on the *filepointer* relative position within the *file*. If *filepointer* is zero, the current record pointer is cleared and no record is retrieved.

GET(*key,keypointer*)

Gets a record from the file based on the *keypointer* relative position within the *key*.

The values for *filepointer* and *keypointer* are file driver dependent. They could be: record number; relative byte position within the file; or, some other kind of "seek position" within the file. If the *filepointer* or *keypointer* value is out of range, or there are no matching *key* values in the data file, the "Record Not Found" error is posted.

The DUPLICATE procedure assumes that the contents of the RECORD structure data buffer are duplicated at the current record pointer location. Therefore, when using DUPLICATE prior to ADDing a record, the record pointer should be cleared with: GET(*file,0*).

QUEUE Usage

GET reads an entry into the QUEUE structure data buffer for processing. If GET does not find a match, the "Entry Not Found" error is posted.

GET(*queue,pointer*)

Retrieves the entry at the relative entry position specified by the *pointer* value in the order the QUEUE entries were added, or last SORTed. If *pointer* is zero, the value returned by the POINTER procedure is set to zero.

GET(*queue,key*)

Searches for the first QUEUE entry that matches the value in the *key* field(s). Multiple *key* parameters may be used (up to 16), separated by commas. If the QUEUE has not been SORTed on the field(s) used as the *key* parameter(s), the *key* indicates an "alternate sort order" which is then cached (making a subsequent SORT on those same fields very efficient).

GET(queue,name)

Searches for a QUEUE entry that matches the value in the *name* field(s). The *name* string must contain the NAME attributes of the fields, separated by commas, with optional leading plus or minus signs to indicate ascending or descending sequence. If the QUEUE has not been SORTed on the *named* field(s), the *name* indicates an "alternate sort order" which is then cached (making a subsequent SORT on those same fields very efficient).

GET(queue,function)

GET by FUNCTION will read from a positional value returned by the function. See Additional Queue Considerations.

Errors Posted: 08 Insufficient Memory
 30 Entry Not Found
 35 Record Not Found
 37 File Not Open
 43 Record Is Already Held
 75 Invalid Field Type Descriptor

Example:

```
NameQue  QUEUE,PRE(Que)
Name     STRING(20),NAME('FirstField')
Zip      DECIMAL(5,0),NAME('SecondField')
        END

Customer FILE,DRIVER('Clarion'),PRE(Cus)
NameKey  KEY(Cus:Name),OPT
NbrKey   KEY(Cus:Number),OPT
Rec      RECORD
Name     STRING(20)
Number   SHORT
        END
        END

CODE
DO BuildQue           !Call routine to build the queue
GET(NameQue,1)       !Get the first entry
IF ERRORCODE() THEN STOP(ERROR()) END

Que:Name = 'Jones'   !Initialize key field
GET(NameQue,Que:Name) !Get the matching record

IF ERRORCODE()
  STOP(ERROR())
END

Que:Name = Fil:Name   !Initialize to value in Fil:Name
GET(NameQue,Que:Name) !Get the matching record
IF ERRORCODE() THEN STOP(ERROR()) END
```

```
Que:Name = 'Smith'           !Initialize the key fields
Que:Zip = 12345
GET(NameQue,'FirstField,SecondField') !Get the matching record
IF ERRORCODE() THEN STOP(ERROR()) END

LOOP X# = 1 TO RECORDS(NameQue)
  GET(NameQue,X#)           !Loop through every entry in the QUEUE
  IF ERRORCODE() THEN STOP(ERROR()) END
  !Process the entry
END

Cus:Name = 'Clarion'        !Initialize key field
GET(Customer,Cus:NameKey)   !get record with matching value
IF ERRORCODE() THEN STOP(ERROR()) END

GET(Customer,3)            !Get 3rd rec in physical file order
IF ERRORCODE() THEN STOP(ERROR()) END

GET(Cus:NameKey,3)         !Get 3rd rec in keyed order
IF ERRORCODE() THEN STOP(ERROR()) END
```

See Also:

SORT

PUT

POINTER(queue)

POINTER

DUPLICATE

HOLD

WATCH

GETFONT (get font information)

GETFONT(*control* [, *typeface*] [, *size*] [, *color*] [, *style*] [, *charset*])

GETFONT	Gets display font information.
<i>control</i>	A field number or field equate label for the control from which to get the information. If <i>control</i> is zero (0), it specifies the WINDOW.
<i>typeface</i>	A string variable to receive the name of the font.
<i>size</i>	An integer variable to receive the size (in points) of the font.
<i>color</i>	A LONG integer variable to receive the red, green, and blue values for the color of the font in the low-order three bytes. If the value is negative, the <i>color</i> represents a system color.
<i>style</i>	An integer variable to receive the strike weight and style of the font.
<i>charset</i>	A LONG integer variable to receive the character set value.

GETFONT gets the display font information for the *control*. If the *control* parameter is zero (0), **GETFONT** gets the default display font for the window.

Example:

```
TypeFace  STRING( 31 )
Size      BYTE
Color     LONG
Style     LONG
CharSet   LONG

CODE
OPEN( Screen )
GETFONT( 0, TypeFace, Size, Color, Style, CharSet)    !Get font info for the window
```

See Also:

SETFONT

GETINI (return INI file entry)

GETINI(*section* ,*entry* [,*default*] [,*file*])

GETINI	Returns the value for an INI file entry.
<i>section</i>	A string constant or variable containing the name of the portion of the INI file which contains the <i>entry</i> .
<i>entry</i>	A string constant or variable containing the name of the specific setting for which to return the value.
<i>default</i>	A string constant or variable containing the default value to return if the <i>entry</i> does not exist (up to 1023 characters). If omitted and the entry does not exist, GETINI returns an empty string.
<i>file</i>	A string constant or variable containing the name of the INI file to search (looks for the <i>file</i> in the Windows directory unless a full path is specified). If the <i>file</i> does not exist, GETINI returns an empty string. If omitted, GETINI searches the WIN.INI file.

The **GETINI** procedure returns the value of an *entry* in a Windows-standard INI file (maximum file size is 64K). A Windows-standard INI file is an ASCII text file with the following format:

```
[some section name]
entry=value
next entry=another value
```

For example, WIN.INI contains entries such as:

```
[intl]
sLanguage=enu
sCountry=United States
iCountry=1
```

The GETINI procedure searches the specified *file* for the *entry* within the *section* you specify. It returns everything on the *entry*'s line of text that appears to the right of the equal sign (=).

Return Data Type: **STRING**

Example:

```
Value STRING(30)
CODE
Value = GETINI('intl','sLanguage')    !Get the language entry
```

See Also:

PUTINI

GETNULLS(get the NULL state of a table)

GETNULLS(*table*)

GETNULLS Gets the NULL state of a table.

table The label of a FILE.

The **GETNULLS** procedure returns a string containing the NULL state of the table. The size of the return string is file driver dependent. For the AS400 Accelerator, MSSQL Accelerator, ODBC Accelerator, Oracle Accelerator, Scalable/Pervasive Accelerator, and SQLAnywhere Accelerator the size of the return string is 4 times the number of columns in the table. This function is valid on all SQL tables.

Return Data Type: **STRING**

Example:

```
StorageString STRING(255)
Rec           STRING(2048)
```

CODE

```
StorageString = GETNULLS(table) !Get NULL state of table
Rec = table.record             !Save record buffer
! Do Table Processing
table.record = Rec             !Restore record buffer
SETNULLS(table, StorageString) !Set NULL state of table columns
```

Errors Posted: 80 Function Not Supported

GETPOSITION (get control position)

GETPOSITION(*control* [, *x*] [, *y*] [, *width*] [, *height*])

GETPOSITION Gets the position and size of an APPLICATION, WINDOW, or control.

<i>control</i>	A field number or field equate label for the control from which to get the information. If <i>control</i> is zero (0), it specifies the window.
<i>x</i>	An integer variable to receive the horizontal position of the top left corner.
<i>y</i>	An integer variable to receive the vertical position of the top left corner.
<i>width</i>	An integer variable to receive the width.
<i>height</i>	An integer variable to receive the height.

GETPOSITION gets the position and size of an APPLICATION, WINDOW, or control. The position and size values are dependent upon the presence or absence of the SCROLL attribute on the *control*. If SCROLL is present, the values are relative to the virtual window. If SCROLL is not present, the values are relative to the top left corner of the currently visible portion of the window. This means the values returned always match those specified in the AT attribute or most recent SETPOSITION.

The values in the *x*, *y*, *width*, and *height* parameters are measured in dialog units. Dialog units are defined as one-quarter the average character width by one-eighth the average character height. The size of a dialog unit is dependent upon the size of the default font for the window. This measurement is based on the font specified in the FONT attribute of the window, or the system default font specified by Windows.

Example:

```
Screen WINDOW,PRE(Scr)
    ENTRY(@N3),USE(Ctl:Code)
    ENTRY(@S30),USE(Ctl:Name)
    BUTTON('OK'),USE(?OkButton),KEY(EnterKey)
    BUTTON('Cancel'),USE(?CanxButton),KEY(EscKey)
END

X      SHORT
Y      SHORT
Width  SHORT
Height SHORT
CODE
OPEN(Screen)
GETPOSITION(?Ctl:Code,X,Y,Width,Height)
```

See Also: SETPOSITION

GETREG(get Windows registry entry)

GETREG(LONG *root*, STRING *keyname* [, STRING *valuename*]),STRING

GETREG Gets the value of a specific key and/or value from the system registry.

root The root section of the registry from which to obtain the value. Valid values for this are defined in equates.clw and are as follows:

REG_CLASSES_ROOT
REG_CURRENT_USER
REG_LOCAL_MACHINE
REG_USERS
REG_PERFORMANCE_DATA
REG_CURRENT_CONFIG
REG_DYN_DATA

keyname The key name of the key whose value is to be queried. This may contain a path separated by backslash '\' characters.

valuename The name of the value to be queried, if omitted, the value associated directly with the key is returned.

The **GETREG** function returns the value of named entry in the system registry as a Clarion string. If the requested entry does not exist, an empty string is returned.

Return Data Type: **STRING**

Example:

```

PROGRAM

MAP.
  INCLUDE('EQUATES')
  CurrentPath CSTRING(100)
  ColorScheme CSTRING(100)

CODE
  CurrentPath =|
  GETREG(REG_LOCAL_MACHINE,'SOFTWARE\SoftVelocity\Clarion6','root')
  !Returns root directory of Clarion 6 install

  ColorScheme =|
  GETREG(REG_CURRENT_USER,'Control Panel\Current','Color Schemes')
  !get the current user's color scheme

```

See Also: **PUTREG, DELETEREG**

GETSTATE (return current state of data file)

GETSTATE(*file* [,*saveblob*])

GETSTATE	Identifies a file's current state including the record buffer, file pointer, file sequencing order, and the error state of the system.
<i>file</i>	The label of a FILE declaration.
<i>saveblob</i>	An integer constant or variable containing either zero (0) or one (1). If one, the state of the BLOB is also returned. If omitted or zero (0) the BLOB state is not returned.

GETSTATE returns the current state of the data file. This includes the record buffer, MEMOs, BLOBs, file pointer, file sequencing order, and the current error state of the system. Use **GETSTATE** to store the file's state so it can be restored using **RESTORESTATE**. This set of functions, (including **FREESTATE**), allow for easy interruption and resumption of sequential file processes.

GETSTATE can now save the state of BLOBs if the second parameter is set to TRUE. It defaults to not saving BLOBs.

Return Data Type: LONG

Example:

```
MyFunction PROCEDURE(FILE MyFile)
```

```
CurState LONG
```

```
CODE
```

```
!File will be in the same state on exit of this procedure as it was on entry
```

```
CurState = GETSTATE(MyFile,1)      !save current state of file and BLOB
SET(MyFile)                        !ready to access file
NEXT(MyFile)                        !read a record
CLEAR(MyFile)                       !clear record buffer
RESTORESTATE(MyFile, CurState)      !restore file to initial state
FREESTATE(MyFile, CurState)         !release resources
```

See Also:

FREESTATE

RESTORESTATE

HALT (exit program)

HALT([*errorlevel*] [,*message*])

HALT	Immediately terminates the program.
<i>errorlevel</i>	A positive integer constant or variable (range: 0 - 250) which is the exit code to pass to DOS, setting the DOS ERRORLEVEL. If omitted, the default is zero.
<i>message</i>	A string constant or variable which is typed on the screen after program termination.

The **HALT** statement immediately returns to the operating system, setting the *errorlevel* and optionally displaying a *message* after the program terminates. All standard runtime library procedures for application closedown are performed (all open windows and files are closed and flushed and all allocated memory is returned to the operating system) without executing any further Clarion code in the application.

Example:

```

PasswordProc PROCEDURE
Password  STRING(10)
Window  WINDOW,CENTER
        ENTRY(@s10),AT(5,5),USE(Password),HIDE
        END
CODE
OPEN(Window)
ACCEPT
CASE ACCEPTED( )
OF ?Password)
    IF Password <> 'Pay$MeMoRe'
        HALT(0,'Incorrect Password entered.')
    END
END
END
END

```

See Also:

STOP

HELP (help window access)

HELP([*helpfile*] [,*window-id*])

HELP	Opens a help file and activates a help window.
<i>helpfile</i>	A string constant or the label of a STRING variable that has the DOS directory file specification for the help file. If the file specification does not contain a complete path and filename, the help file is assumed to be in the current directory. If the file extension is omitted, ".HLP" is assumed. If the <i>helpfile</i> parameter is omitted, a comma is required to hold its position.
<i>window-id</i>	A string constant or the label of a STRING variable that contains the key used to access the help system. This may be either a help keyword or a "context string."

The **HELP** statement opens a designated *helpfile*, and activates the window named by the *window-id*. While an ASK or ACCEPT is controlling program execution, the active help window is displayed when the operator presses F1 (the "Help" key).

If the *window-id* parameter is omitted, the *helpfile* is nominated but not opened. If the *helpfile* parameter is omitted, the current help file is opened, and the window identified by *window-id* is activated. If both parameters are omitted, the current *helpfile* is opened at the current topic.

The *window-ID* may contain a Help keyword. This is a keyword that is displayed in the Help Search dialog. When the user presses F1, if only one topic in the help file specifies this keyword, the help file is opened at that topic; if more than one topic specifies the keyword, the search dialog is opened for the user.

A "context string" is identified by a leading tilde (~) in the *window-ID*, followed by a unique identifier associated with exactly one help topic. If the tilde is missing, the *window-ID* is assumed to be a help keyword. When the user presses F1, the help file is opened at the specific topic associated with that "context string."

Newly started threads inherit the help file settings from the previous thread that started it.

Example:

```
HELP('C:\HLPDIR\LEDGER.HLP')  !Open the gen ledger help file
HELP(, '~CustUpd')            !Activate customer update help window
HELP                          !Display the help window
```

See Also:

ASK

ACCEPT

HLP

HIDE (blank a control)

HIDE([*first control*] [, *last control*])

HIDE Hides window controls.

first control Field number or field equate label of a control, or the first control in a range of controls. If omitted, defaults to zero (0).

last control Field number or field equate label of the last control in a range of controls.

The **HIDE** statement hides a control, or range of controls, on an APPLICATION or WINDOW structure. When hidden, the control does not appear on screen.

Example:

```
Screen WINDOW,PRE(Scr)
    ENTRY(@N3),USE(Ctl:Code)
    ENTRY(@S30),USE(Ctl:Name)
    BUTTON('OK'),USE(?OkButton),KEY(EnterKey)
    BUTTON('Cancel'),USE(?CanxButton),KEY(EscKey)
END

CODE
OPEN(Screen)
HIDE(?Ctl:Code)           !Hide a control
HIDE(?Ctl:Code,?Ctl:Name) !Hide range of controls
HIDE(2)                   !Hide the second control
```

See Also:

UNHIDE

ENABLE

DISABLE

HOLD (exclusive record access)

HOLD(*entity* [,*seconds*])

HOLD	Arms record locking.
<i>entity</i>	The label of a FILE opened for shared access or a VIEW whose component files are opened for shared access.
<i>seconds</i>	A numeric constant or variable which specifies the maximum wait time in seconds.

The **HOLD** statement arms record locking for a following GET, REGET, NEXT, or PREVIOUS statement in a multi-user environment. The GET, REGET, NEXT, or PREVIOUS flags the record as "held" when it successfully gets the record. Generally, this excludes other users from writing to, but not reading, the record. The specific action HOLD takes is file driver dependent. When the *entity* parameter is the label of a VIEW structure, HOLD operates on the primary file in the VIEW, only.

HOLD(*entity*)

Arms HOLD so that the following GET, REGET, NEXT, or PREVIOUS attempts to hold the record until it is successful. If it is held by another workstation, GET, REGET, NEXT, or PREVIOUS will wait until the other workstation releases it.

HOLD(*entity* , *seconds*)

Arms HOLD for the following GET, REGET, NEXT, or PREVIOUS to post the "Record Is Already Held" error after unsuccessfully trying to hold the record for *seconds*.

A user may only HOLD one record at a time. If a second record is to be accessed in the same file, the previously held record must be released (see RELEASE).

A common problem to avoid is "deadly embrace." This occurs when two workstations attempt to hold the same set of records in two different orders and both are using the **HOLD**(*entity*) form of HOLD. One workstation has already held a record that the other is trying to HOLD, and vice versa. You can avoid this problem by using the **HOLD**(*entity,seconds*) form of HOLD, and trapping for the "Record Is Already Held" error after the GET, REGET, NEXT, or PREVIOUS statement.

Example:

```

ViewOrder VIEW(Customer) !Declare VIEW structure
    PROJECT(Cus:AcctNumber,Cus:Name)
    JOIN(Hea:AcctKey,Cus:AcctNumber) !Join Header file
    PROJECT(Hea:OrderNumber)
    JOIN(Dtl:OrderKey,Hea:OrderNumber) !Join Detail file
    PROJECT(Det:Item,Det:Quantity)
    JOIN(Pro:ItemKey,Dtl:Item) !Join Product file
    PROJECT(Pro:Description,Pro:Price)
    END
    END
    END
    END

CODE
OPEN(Customer,22h)
OPEN((Header,22h)
OPEN(Detail,22h)
OPEN(Product,22h)
SET(Cus:AcctKey)
OPEN(ViewOrder)
    LOOP !Process records Loop
        LOOP !Loop to avoid "deadly embrace"
            HOLD(ViewOrder,1) !Arm Hold on view, try for 1 second
            NEXT(ViewOrder) !Get and hold the record
            IF ERRORCODE() = 43 !If someone else has it
                CYCLE ! try again
            ELSE
                BREAK !Break if not held
            END
        END
    END
    IF ERRORCODE() THEN BREAK END !Check for end of file
    !Process the records
    RELEASE(ViewOrder) !release held records
    END
CLOSE(ViewOrder)

```

See Also:

RELEASE

NEXT

PREVIOUS

WATCH

GET

REGET

IDLE (arm periodic procedure)

IDLE([*procedure*] [,*separation*])

IDLE	Arms a <i>procedure</i> that periodically executes.
<i>procedure</i>	The label of a PROCEDURE. The <i>procedure</i> may not take any parameters.
<i>separation</i>	An integer that specifies the minimum wait time (in seconds) between calls to the <i>procedure</i> . A <i>separation</i> of 0 specifies continuous calls. If <i>separation</i> is omitted, the default value is 1 second.

An **IDLE** procedure is active while ASK or ACCEPT are waiting for user input. Only one IDLE procedure may be active at a time. Naming a new IDLE *procedure* overrides the previous one. An IDLE statement with no parameters disarms the IDLE process.

The IDLE *procedure* executes on thread one (1)--the same thread as the APPLICATION frame in an MDI application. Therefore, any WINDOW structure in an IDLE *procedure* must not have the MDI attribute. Since opening a non-MDI window in the same thread as the APPLICATION frame creates an application modal window, it would be more usual for an IDLE *procedure* not to have a WINDOW structure at all.

An IDLE *procedure* is usually prototyped in the PROGRAM's MAP. If prototyped in a MEMBER MAP, the IDLE statements which activate and de-activate it must be contained in a procedure within the same MEMBER module.

Example:

```
IDLE(ShoTime,10)      !Call shotime every 10 seconds
IDLE(CheckNet)       !Check network activity every 1 second
IDLE                  !Disarm idle procedure
```

See Also:

ASK

ACCEPT

PROCEDURE

MAP

MDI

IMAGE (draw a graphic image)

IMAGE(*x* , *y* , [*width*] , [*height*] , *filename* [, *attributelist*])

IMAGE	Places a graphic image on the current window or report.
<i>x</i>	An integer expression that specifies the horizontal position of the starting point.
<i>y</i>	An integer expression that specifies the vertical position of the starting point.
<i>width</i>	An integer expression that specifies the width. This may be a negative number. If omitted, defaults to the width of the graphic as it is stored.
<i>height</i>	An integer expression that specifies the height. This may be a negative number. If omitted, defaults to the height of the graphic as it is stored.
<i>filename</i>	A string constant or variable containing the name of the file to display.
<i>attributelist</i>	A string constant, variable, or EQUATE containing an optional type of output document and its associated attributes. Only valid when the target is a REPORT. (See <i>EXTEND</i>)

The **IMAGE** procedure places a graphic image on the current window or report at the position and size specified by its *x*, *y*, *width*, and *height* parameters. This may be a bitmap (.BMP), icon (.ICO), PaintBrush (.PCX), Graphic Interchange Format (.GIF), JPEG (.JPG), or Windows metafile (.WMF).

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
    !window controls
    END
    CODE
    OPEN(MDIChild)
    IMAGE(100,50,100,50,'LOGO.BMP')    !Draw graphic image
```

See Also:

Current Target

SETPENCOLOR

SETPENWIDTH

SETPENSTYLE

INCOMPLETE (return empty REQ control)

INCOMPLETE()

The **INCOMPLETE** procedure returns the field number of the first control with the REQ attribute in the currently active window that has been left zero or blank, and gives input focus to that control. If all REQ controls in the window contain data, INCOMPLETE returns zero (0) and leaves input focus on the control that already had it.

The INCOMPLETE procedure duplicates the action performed by the REQ attribute on a BUTTON control.

Return Data Type: SIGNED

Example:

```
CODE
OPEN(Screen)
ACCEPT
CASE ACCEPTED()
OF ?OkButton
  IF INCOMPLETE()      !Any REQ fields empty?
    SELECT(INCOMPLETE()) ! if so, go to it
    CYCLE
  ELSE
    BREAK              !If not, go on
  END
END
END
END
```

See Also:

REQ

BUTTON

INLIST (return entry in list)

INLIST(*searchstring*,*liststring*,*liststring* [,*liststring*...])

INLIST Returns item in a list.

searchstring A constant, variable, or expression that contains the value for which to search. If the value is numeric, it is converted to a string before comparisons are made.

liststring The label of a variable or constant value to compare against the *searchstring*. If the value is numeric, it is converted to a string before comparisons are made. There may be up to 25 *liststring* parameters, and there must be at least two.

The **INLIST** procedure compares the contents of the *searchstring* against the values contained in each *liststring* parameter. If a matching value is found, the procedure returns the number of the first *liststring* parameter containing the matching value (relative to the first *liststring* parameter). If the *searchstring* is not found in any *liststring* parameter, **INLIST** returns zero.

Return Data Type: LONG

Example:

```
!INLIST('D','A','B','C','D','E') returns 4
!INLIST('B','A','B','C','D','E') returns 2
```

```
EXECUTE INLIST(Emp:Status,'Fulltime','Parttime','Retired','Consultant')
  Scr:Message = 'All Benefits'           !Full timer
  Scr:Message = 'Holidays Only'         !Part timer
  Scr:Message = 'Medical/Dental Only'   !Retired
  Scr:Message = 'No Benefits'           !Consultant
END
```

See Also:

CHOOSE

INRANGE (check number within range)

INRANGE(*expression,low,high*)

INRANGE Return number in valid range.

expression A numeric constant, variable, or expression.

low A numeric constant, variable, or expression of the lower boundary of the range.

high A numeric constant, variable, or expression of the upper boundary of the range.

The **INRANGE** procedure compares a numeric *expression* to an inclusive range of numbers. If the value of the *expression* is within the range, the procedure returns the value 1 for "true." If the *expression* is greater than the *high* parameter, or less than the *low* parameter, the procedure returns a zero for "false."

Return Data Type: LONG

Example:

```
IF INRANGE(Date % 7,1,5) !If this is a week day
  DO WeekdayRate         !use the weekday rate
ELSE
  !Otherwise
  DO WeekendRate        !use the weekend rate
END
```

INSTANCE (return variable's thread instance address)

INSTANCE(*variable*,*threadno*)

INSTANCE Returns the address of a variable or entity's thread instance.

Variable Label of a variable, field, FILE, KEY or QUEUE or reference variable.

threadno A numeric constant, variable, or expression that can be evaluated as a SIGNED integer.

The **INSTANCE** procedure evaluates the condition of the *variable* parameter and its thread number referenced by the *threadno* parameter and returns the following results:

If the value of *threadno* is not zero(0), and the thread referenced by *threadno* is not started (or an instance of *variable* is not allocated for the thread referenced by *threadno*), **INSTANCE** returns zero(0). Otherwise, **INSTANCE** returns the address of the *variable* instance allocated for the active thread referenced by *threadno*.

If the *variable* parameter is not a threaded variable or entity, or the threaded variable or entity is allocated for a thread other than one referenced by the *threadno*, **INSTANCE** returns the ADDRESS (*variable*).

If the *threadno* parameter is set to zero(0), **INSTANCE** returns the address of the *variable* that is assigned by the program loader. This is also known as the variables' *thread independent ID*.

INSTANCE can be used instead of the ADDRESS() statement when ADDRESS() is not valid or available (e.g. FILE and QUEUE structures). ADDRESS(QUEUE) is a legal call, but it returns the address of the queue's internal buffer. On the other hand, **INSTANCE**(QUEUE,THREAD()) returns the address of the queue's internal structure.

For example, given the following QUEUE declarations:

```
SomeQueue QUEUE
...
      END
QueueRef &QUEUE
```

The following assignment is correct:

```
QueueRef &= INSTANCE (SomeQueue, Somethread)
```

while the assignment shown below is not correct:

```
QueueRef &= ADDRESS (SomeQueue)
```

and sets the QueueRef variable to the wrong value.

INSTANCE is also valuable when you need the thread independent ID of the variable.

Return Data Type: LONG

Example:

```
addressvar = INSTANCE(SalesFile,THREAD())
           !return address of SalesFile entity on active thread

addressvar = INSTANCE(GLO:LoginID, 0)
           !get the thread independent ID of a global threaded var
```

INSTRING (return substring position)

INSTRING(*substring*,*string* [,*step*] [,*start*])

INSTRING	Searches for a substring in a string.
<i>substring</i>	A string constant, variable, or expression that contains the string for which to search. You should CLIP a variable <i>substring</i> so INSTRING will not look for a match that contains the trailing spaces in the variable.
<i>string</i>	A string constant, or the label of the STRING, CSTRING, or PSTRING variable to be searched.
<i>step</i>	A numeric constant, variable, or expression which specifies the step length of the search. A <i>step</i> of 1 will search for the <i>substring</i> beginning at every character in the <i>string</i> , a <i>step</i> of 2 starts at every other character, and so on. A negative <i>step</i> value will search from right to left within the <i>string</i> . If <i>step</i> is omitted, the step length defaults to the length of the <i>substring</i> .
<i>start</i>	A numeric constant, variable, or expression which specifies where to begin the search of the <i>string</i> . If omitted, the search starts at the first character position.

The **INSTRING** procedure *steps* through a *string*, searching for the occurrence of a *substring*. If the *substring* is found, the procedure returns the *step* number on which the *substring* was found. If the *substring* is not found in the *string*, **INSTRING** returns zero.

INSTRING starts to search for *substring* from the *start* position in the *string* and moves forward with *step* until the *substring* is found, or the unchecked tail of the *string* is less than length of the *substring*. In the latter case, **INSTRING** returns zero. If the *substring* is found, the result is equal to the number of *steps* from the origin of the *string* to the found position. If the value of *step* is not equal to 1, the result is rounded up to the whole number of steps as follows:

$$\text{INT} ((\text{found position} - 1) / \text{step}) + 1$$

Return Data Type: **UNSIGNED**

Example:

```

INSTRING('DEF','ABCDEFGHIJ',1,1) !returns 4
INSTRING('DEF','ABCDEFGHIJ',1,2) !returns 4
INSTRING('DEF','ABCDEFGHIJ',1,3) !returns 4
INSTRING('DEF','ABCDEFGHIJ',1,4) !returns 4
INSTRING('DEF','ABCDEFGHIJ',1,5) !returns 0

INSTRING('DEF','ABCDEFGHIJ',2,1) !returns 0
INSTRING('DEF','ABCDEFGHIJ',2,2) !returns 2
INSTRING('DEF','ABCDEFGHIJ',3,1) !returns 2

```

```
!Extract extension from file spec:
Extension = SUB(FileSpec,INSTRING('.',FileSpec) + 1,3)

IF INSTRING(CLIP(Search),Cus:Notes,1,1) !If search variable found
  Scr:Message = 'Found' !display message
END
```

See Also:

SUB, STRING, CSTRING, PSTRING, String Slicing

INT (truncate fraction)

INT(*expression*)

INT Return integer.

expression A numeric constant, variable, or expression.

The **INT** procedure returns the integer portion of a numeric expression. No rounding is performed, and the sign remains unchanged.

Return Data Type: REAL or DECIMAL

Example:

```
!INT(8.5)        returns 8
!INT(-5.9)      returns -5
```

```
x = INT(y)        !Return integer portion of y variable contents
```

See Also:

BCD Operations and Procedures

ROUND

ISALPHA (return alphabetic character)

ISALPHA(*string*)

ISALPHA Returns whether the *string* passed to it contains an alphabetic character.

string The label of the character string to test. If the *string* contains more than one character, only the first character is tested.

The ISALPHA procedure returns TRUE if the *string* passed to it is alphabetic (an upper or lower case letter) and false otherwise. This is independent of the language and collation sequence. This procedure requires that CLACASE has been set in the application's environment file or through the LOCALE statement.

Return Data Type: LONG

Example:

```
SomeString STRING(1)
CODE
SomeString = 'A'           !ISALPHA returns true
IF ISALPHA(SomeString)
  X#= MESSAGE('Alpha string')
END
SomeString = '1'           !ISALPHA returns false
IF ISALPHA(SomeString)
  X#= MESSAGE('Alpha string')
ELSE
  X#= MESSAGE('Not Alpha string')
END
```

See Also:

ISUPPER

ISLOWER

LOCALE

Environment Files

ISLOWER (return lower case character)

ISLOWER(*string*)

ISLOWER	Returns whether the <i>string</i> passed to it contains a lower case alphabetic character.
<i>string</i>	The label of the string to test. If the <i>string</i> contains more than one character, only the first character is tested.

The ISLOWER procedure returns TRUE if the *string* passed to it is a lower case letter and false otherwise. This is independent of the language and collation sequence. This procedure requires that CLACASE has been set in the application's environment file or through the LOCALE statement.

Return Data Type: LONG

Example:

```
SomeString STRING(1)
CODE
SomeString = 'a'           !ISLOWER returns true
IF ISLOWER(SomeString)
  X#= MESSAGE('Lower case string')
END
SomeString = 'A'           !ISLOWER returns false
IF ISLOWER(SomeString)
  X#= MESSAGE('Lower case string')
ELSE
  X#= MESSAGE('Not lower case string')
END
```

See Also:

ISUPPER

ISALPHA

LOCALE

Environment Files

ISSTRING (return field string type or not)

ISSTRING(*field*)

ISSTRING Returns true if the *field* is a STRING, CSTRING, or PSTRING data type.

field The label of a field.

The **ISSTRING** statement returns true if the *field* is a STRING, CSTRING, or PSTRING data type.

Return Data Type: SIGNED

Example:

```
MyGroup  GROUP
F1       LONG      !Field number 1
F2       SHORT     !Field number 2
F3       STRING(30) !Field number 3
InGroup  GROUP     !Field number 3
F1       LONG      !Field number 4
F2       SHORT     !Field number 5
F3       STRING(30) !Field number 6
          END
          END

Flag LONG
CODE
Flag = ISSTRING(MyGroup.F1) !returns FALSE

Flag = ISSTRING(MyGroup.F3) !returns TRUE
```

See Also:

WHAT

WHERE

ISUPPER (return upper case character)

ISUPPER(*string*)

ISUPPER	Returns whether the <i>string</i> passed to it contains an upper case alphabetic character.
<i>string</i>	The label of the string to test. If the <i>string</i> contains more than one character, only the first character is tested.

The ISUPPER procedure returns TRUE if the *string* passed to it is an upper case letter and false otherwise. This is independent of the language and collation sequence. This procedure requires that CLACASE has been set in the application's environment file or through the LOCALE statement.

Return Data Type: LONG

Example:

```
SomeString STRING(1)
CODE
SomeString = 'A'           !ISUPPER returns true
IF ISUPPER(SomeString)
  X#= MESSAGE('Upper case string')
END
SomeString = 'a'           !ISUPPER returns false
IF ISUPPER(SomeString)
  X#= MESSAGE('Upper case string')
ELSE
  X#= MESSAGE('Not upper case string')
END
```

See Also:

ISLOWER

ISALPHA

LOCALE

Environment Files

KEYBOARD (return keystroke waiting)

KEYBOARD()

The **KEYBOARD** procedure returns the keycode of the first keystroke in the keyboard buffer. It is used to determine if there are keystrokes waiting to be processed by an ASK or ACCEPT statement.

Return Data Type: UNSIGNED

Example:

```
LOOP UNTIL KEYBOARD()           !Wait for any key
  ASK
  IF KEYCODE() = EscKey THEN BREAK END  !On esc key, break the loop
END
```

See Also:

ASK

ACCEPT

KEYCODE

Keycode Equate Labels

KEYCHAR (return ASCII code)

KEYCHAR()

The **KEYCHAR** procedure returns the ASCII value of the last key pressed at the time the event occurred.

Return Data Type: UNSIGNED

Example:

```
ACCEPT                                !Wait for an event
CASE KEYCHAR()                       !Process the last keystroke
OF VAL('A') TO VAL('Z') ! upper case?
  DO ProcessUpper
OF VAL('a') TO VAL('z') ! lower case?
  DO ProcessLower
END
END
```

See Also:

SETKEYCHAR

ASK

ACCEPT

SELECT

VAL

CHR

KEYCODE (return last keycode)

KEYCODE()

The **KEYCODE** procedure returns the keycode of the last key pressed at the time the event occurred, or the last keycode value set by the SETKEYCODE procedure.

Return Data Type: UNSIGNED

Example:

```
ACCEPT                !Loop on the display
CASE KEYCODE()        !Process the keystroke
OF UpKey              !up arrow
  DO GetRecordUp      !get a record
OF DownKey            !down arrow
  DO GetRecordDn      !get a record
END
END
```

See Also:

ASK

ACCEPT

KEYBOARD

SETKEYCODE

KEYSTATE

Keycode Equate Labels

KEYSTATE (return keyboard status)

KEYSTATE()

The **KEYSTATE** procedure returns a bitmap containing the status of the SHIFT, CTRL, ALT, any extended key, CAPS LOCK, NUM LOCK, SCROLL LOCK, and INSERT keys for the last KEYCODE procedure return value. The bitmap is contained in the high-order byte of the returned SHORT.

```

x . . . . . insert key (8000h)
. x . . . . . scroll lock (4000h)
. . x . . . . . num lock (2000h)
. . . x . . . . . caps lock (1000h)
. . . . x . . . . . extended (0800h)
. . . . . x . . . . . alt (0400h)
. . . . . . x . . . . . ctrl (0200h)
. . . . . . . x . . . . . shift (0100h)

```

Return Data Type: **UNSIGNED**

Example:

```

ACCEPT                !Loop on the display
CASE KEYCODE()        !Process the keystroke
OF EnterKey           !User pressed Enter
  IF BAND(KEYSTATE(),0800h) !Detect enter on numeric keypad
    PRESSKEY(TabKey)      !press tab for the user
  END
END
END
END

```

See Also:

KEYCODE

BAND

LASTFIELD (return last window control)

LASTFIELD()

The **LASTFIELD** procedure returns the highest field number in the currently active window (or REPORT) as specified by SETTARGET. This does not include any controls in a TOOLBAR or MENUBAR or any controls created after the window is opened.

Return Data Type: SIGNED

Example:

```
DISABLE(FIRSTFIELD(),LASTFIELD())    !Dim all control fields
```

See Also:

FIRSTFIELD

LEFT (return left justified string)

LEFT(*string* [,*length*])

LEFT	Left justifies a string.
<i>string</i>	A string constant, variable, or expression.
<i>length</i>	A numeric constant, variable, or expression for the length of the return string. If omitted, <i>length</i> defaults to the length of the <i>string</i> .

The **LEFT** procedure returns a left justified string. Leading spaces are removed from the *string*.

Return Data Type: **STRING**

Example:

```
!LEFT(' ABC') returns 'ABC '
```

```
CompanyName = LEFT(CompanyName) !Left justify the company name
```

See Also:

RIGHT

CENTER

LEN (return length of string)

LEN(*string*)

LEN Returns length of a string.

string A string constant, variable, or expression.

The **LEN** procedure returns the length of a *string*. If the *string* parameter is the label of a STRING variable, the procedure will return the declared length of the variable. If the *string* parameter is the label of a CSTRING or PSTRING variable, the procedure will return the length of the contents of the variable. Numeric variables are automatically converted to STRING intermediate values.

Return Data Type: UNSIGNED

Example:

```
IF LEN(CLIP(Title) & ' ' & CLIP(First) & ' ' & CLIP>Last))>30 !If full name won't fit
  Rpt:Name = CLIP(Title) & ' ' & SUB(First,1,1) & '. ' & Last !use first initial
ELSE
  Rpt:Name = CLIP(Title) & ' ' & CLIP(First) & ' ' & CLIP>Last)!else use full name
END

Rpt:Title = CENTER(Cus:Name,LEN(Rpt:Title))    !Center the name in the title
```

LINE (draw a straight line)

LINE(*x* , *y* , *width* , *height* [, *attributelist*])

LINE	Draws a straight line on the current window or report.
<i>x</i>	An integer expression specifying the horizontal position of the starting point.
<i>y</i>	An integer expression specifying the vertical position of the starting point.
<i>width</i>	An integer expression specifying the width. This may be a negative number.
<i>height</i>	An integer expression specifying the height. This may be a negative number.
<i>attributelist</i>	A string constant, variable, or EQUATE containing an optional type of output document and its associated attributes. Only valid when the target is a REPORT. (See <i>EXTEND</i>)

The **LINE** procedure places a straight line on the current window or report. The starting position, slope, and length of the line are specified by *x*, *y*, *width*, and *height* parameters. The *x* and *y* parameters specify the starting point of the line. The *width* and *height* parameters specify the horizontal and vertical distance to the end point of the line. If these are both positive numbers, the line slopes to the right and down from its starting point. If the *width* parameter is negative, the line slopes left; if the *height* parameter is negative, the line slopes left. If either the *width* or *height* parameter is zero, the line is horizontal or vertical.

<u>Width</u>	<u>Height</u>	<u>Result</u>
positive	positive	right and down from start point
negative	positive	left and down from start point
positive	negative	right and up from start point
negative	negative	left and up from start point
zero	positive	vertical, down from start point
zero	negative	vertical, up from start point
positive	zero	horizontal, right from start point
negative	zero	horizontal, left from start point

The line color is the current pen color set by **SETPENCOLOR**; the default color is the Windows system color for window text. The width is the current width set by **SETPENWIDTH**; the default width is one pixel. The line's style is the current pen style set by **SETPENSTYLE**; the default style is a solid line.

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
    !window controls
END

CODE
OPEN(MDIChild)
LINE(100,50,100,50)    !Draw line
```

See Also: **Current Target**, **SETPENCOLOR**, **SETPENWIDTH**, **SETPENSTYLE**

LOCALE (load environment file)

```
LOCALE( | file          | )
        | setting, value |
```

LOCALE	Allows the user to load a specific environment file (.ENV) at run-time and also to set individual environment settings.
<i>file</i>	A string constant or variable containing the name (including extension) of the environment file (.ENV) to load, or the keyword WINDOWS. This may be a fully-qualified DOS pathname.
<i>setting</i>	A string constant or variable containing the name of the environment variable to set. Valid choices are listed under the <i>Environment Files</i> section.
<i>value</i>	A string constant or variable containing the environment variable setting.

The **LOCALE** procedure allows the user to load a specific environment file (.ENV) at run-time and also to set individual environment settings. This allows an application to load another file to override the default *appname*.ENV file, or to specify individual environment file settings when no environment file exists.

The WINDOWS keyword as the *file* parameter specifies use of Windows' default values for CLACOLSEQ, CLACASE and CLAAMP. When specifying individual *settings*, the *value* parameter does not require double quotes around each individual item in the *value* string, unlike the syntax required in an .ENV file.

Errors Posted:

02	File Not Found
05	Access Denied

Example:

```
LOCALE('MY.ENV')           !Load an environment file
LOCALE('WINDOWS')         !Set default CLACOLSEQ, CLACASE and CLAAMP
LOCALE('CLABUTTON', 'OK,&Si,&No,&Abortar,&Ignora,&Volveratratar,Cancelar,&Ayuda')
                           !Set CLABUTTON to Spanish
LOCALE('CLACOLSEQ', |
'AÅÄäåäåäåæBbCcçDdEēēēēēēFfGgHhIiïïïïJjKkLlMmNññOöøöøPpQqRrSsSttUüüüüüüVvWwXxYyZzÿ')
                           !Set the collating sequence
LOCALE('CLACASE', 'ÅÄÇÉÑÖ,äåçéñö') !Set upper/lower case pairs
LOCALE('CLAMSG2', 'No File Found')  !Set ERROR() message for ERRORCODE()=2
```

See Also: Environment Files, OEM

LOCK (exclusive file access)

LOCK(*file* [,*seconds*])

LOCK	Locks a data file.
<i>file</i>	The label of a FILE opened for shared access.
<i>seconds</i>	A numeric constant or variable which specifies the maximum wait time in seconds.

The **LOCK** statement locks a *file* against access by other workstations in a multi-user environment. Generally, this excludes other users from writing to or reading from the *file*. The file driver may or may not treat separate execution threads within a single program as another workstation or not.

LOCK(*file*)

Attempts to lock the *file* until it is successful. If it is already locked by another workstation, **LOCK** will wait until the other workstation unlocks it.

LOCK(*file,seconds*)

Posts the "File Is Already Locked" error after unsuccessfully trying to lock the file for the specified number of *seconds*.

The most common problem to avoid when locking files is referred to as "deadly embrace." This condition occurs when two workstations attempt to lock the same set of files in two different orders and both are using the **LOCK**(*file*) form of **LOCK**. One workstation has already locked a file that the other is trying to **LOCK**, and vice versa. This problem may be avoided by using the **LOCK**(*file,seconds*) form of **LOCK**, and always locking files in the same order.

Errors Posted: 32 File Is Already Locked

Example:

```

LOOP                !Loop to avoid "deadly embrace"
  LOCK(Master,1)    !Lock the master file, try 1 second
  IF ERRORCODE() = 32 !If someone else has it
    CYCLE          !try again
  END
  LOCK(Detail,1)    !Lock the detail file, try 1 second
  IF ERRORCODE() = 32 !If someone else has it
    UNLOCK(Master)  !unlock the locked file
    CYCLE          !try again
  END
  BREAK            !Break loop when both files are locked
END

```

See Also: UNLOCK, HOLD, LOGOUT

LOCKTHREAD (re-lock the current execution thread)

LOCKTHREAD

The **LOCKTHREAD** statement re-locks the current execution thread that has been unlocked with the **UNLOCKTHREAD** statement.

The **THREADLOCKED()** procedure determines whether the thread has been unlocked or not.

Example:

```
UNLOCKTHREAD           !Unlock the thread
MyLibraryCodeWithMessageLoop !Call the code that has its own message loop
LOCKTHREAD             !Re-lock the thread
```

See Also:

ACCEPT

UNLOCKTHREAD

THREADLOCKED

LOG10 (return base 10 logarithm)

LOG10(*expression*)

LOG10 Returns base 10 logarithm.

expression A numeric constant, variable, or expression. If the value of the *expression* is zero or less, the return value will be zero. The base 10 logarithm is undefined for values less than or equal to zero.

The **LOG10** (pronounced "log ten") procedure returns the base 10 logarithm of a numeric *expression*. The base 10 logarithm of a value is the power to which 10 must be raised to equal that value.

Return Data Type: REAL

Example:

```
!LOG10(10)  returns 1
!LOG10(1)   returns 0
```

```
LogStore = LOG10(Var)           !Store the log 10 of var
```

See Also:

LOGE

LOGE (return natural logarithm)

LOGE(*expression*)

LOGE Returns the natural logarithm.

expression A numeric constant, variable, or expression. If the value of the *expression* is less than zero, the return value is zero. The natural logarithm is undefined for values less than zero.

The **LOGE** (pronounced "log-e") procedure returns the natural logarithm of a numeric *expression*. The natural logarithm of a value is the power to which **e** must be raised to equal that value. The value of **e** used internally by the Clarion library for these calculations is 2.71828182846.

Return Data Type: REAL

Example:

```
!LOGE(2.71828182846) returns 1
!LOGE(1)             returns 0
```

```
LogVal = LOGE(Val)    !Get the natural log of Val
```

See Also:

LOG10

LOGOUT (begin transaction)

LOGOUT(*timeout* [, *file*, ... , *file*])

LOGOUT	Initiates transaction processing.
<i>timeout</i>	A numeric constant or variable specifying the number of seconds to attempt to begin the transaction for a <i>file</i> before aborting the transaction and posting an error.
<i>file</i>	The label of a FILE declaration. There may be multiple <i>file</i> parameters, separated by commas, in the parameter list (up to 100). If no <i>file</i> is specified, all <i>files</i> in the transaction must have been previously named using PROP:Logout.

The **LOGOUT** statement initiates transaction processing for a specified set of *files*. All *files* in the transaction set must have the same file driver and must already be open.

LOGOUT informs the file driver that a transaction is beginning. The file driver then performs the actions necessary to that file system to initiate transaction processing for the specified set of *files*. For example, if the file system requires that the *files* be locked for transaction processing, LOGOUT automatically locks the *files*.

Only one LOGOUT transaction may be active at a time. A second LOGOUT statement without a prior COMMIT or ROLLBACK generates errorcode 56.

Errors Posted: 32 File Is Already Locked
 37 File Not Open
 48 Unable to Log Transaction
 56 LOGOUT Already Active
 80 Function Not Supported

Example:

```
LOGOUT(1,OrderHeader,OrderDetail) !Begin Transaction
DO ErrHandler                      !always check for errors
ADD(OrderHeader)                   !Add Parent record
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                    !always check for errors
  Det:Record = DetailQue           !Assign to record buffer
  ADD(OrderDetail)                 !and add it to the file
  DO ErrHandler                    !always check for errors
END
COMMIT                             !Terminate successful transaction
ASSERT(~ERRORCODE())

ErrHandler ROUTINE                 !Error routine
IF NOT ERRORCODE() THEN EXIT.      !Bail out if no error
Err" = ERROR()                     !Save the error message
ROLLBACK                           !Rollback the aborted transaction
ASSERT(~ERRORCODE())
BEEP                                !Alert the user
MESSAGE('Transaction Error - ' & Err")
RETURN                              !and get out
```

LONGPATH (return long filename)

LONGPATH([*shortfilename*])

LONGPATH Returns the fully-qualified long filename for a given short filename.

shortfilename A string constant, variable, or expression that specifies the DOS standard filename to convert. This may include the complete path. If omitted, **LONGPATH** returns the current drive and directory in long name form.

The **LONGPATH** procedure returns the long filename for a given *short* filename. The file named in the *shortfilename* parameter must already exist on disk. **LONGPATH** can now be used in expressions evaluated by the EVALUATE statements in runtime applications and in the templates.

Return Data Type: **STRING**

Example:

```
MyLongFile STRING(260)
CODE
MyLongFile = LONGPATH('c:\progra~1\mytext~1.txt')
!returns: c:\program files\my text file.txt
```

See Also:

SHORTPATH

PATH

DIRECTORY

LOWER (return lower case)

LOWER(*string*)

LOWER Converts a string to all lower case.

string A string constant, variable, or expression for the *string* to be converted.

The **LOWER** procedure returns a string with all letters converted to lower case.

Return Data Type: **STRING**

Example:

```
!LOWER('ABC') returns 'abc'
```

```
Name = SUB(Name,1,1) & LOWER(SUB(Name,2,19))
```

```
!Make the rest of the name lower case
```

See Also:

UPPER

ISUPPER

ISLOWER

MATCH (return matching values)

MATCH(*first*, *second* [, *mode*])

MATCH	Returns true or false based on a comparison of the first two parameters passed.
<i>first</i>	A string containing data to compare against the <i>second</i> parameter. String constants should be enclosed in single quotes.
<i>second</i>	A string containing data to compare against the <i>first</i> parameter. String constants should be enclosed in single quotes.
<i>mode</i>	An integer constant or equate which specifies the method of comparison. If omitted, a wild card comparison is the default.

The **MATCH** procedure returns true or false as to whether the *first* and *second* parameters match according to the comparison *mode* specified. The following *mode* value EQUATES are listed in EQUATES.CLW:

Match:Simple

A straight-forward equivalence comparison (*first* = *second*), which is most useful when combined with Match:NoCase

Match:Wild (default)

A wild card match with the *second* parameter containing the pattern that can contain asterisk (*) to match 0 or more of any character, and question mark (?) to match any single character.

Match:Regular

A regular expression match where the *second* parameter contains the regular expression. Repeated usage with the same regular expression value is optimized (to avoid recompiling the expresison).

Match:Soundex

A standard soundex comparison of the two strings, returning true if they have the same soundex value.

Match:NoCase

Add to the *mode* for a case insensitive match (except Soundex).

Regular Expression Operators

Regular expressions are used to describe patterns in text. The following characters are regular expression operators (or metacharacters) used to increase the power and versatility of regular expressions.

- ^ Caret matches the beginning of the string or the beginning of a line within the string. For example:

```
^@chapter
```

matches the "@chapter" at the beginning of a string.
- \$ Dollar sign is similar to the caret, but it matches only at the end of a string or the end of a line within the string. For example:

```
p$
```

matches a record that ends with a p.
- .

Period matches any single character except a new line. For example:

```
.P
```

matches any single character followed by a P in a string. Using concatenation we can make regular expressions like 'U.A', which matches any three-character sequence that begins with 'U' and ends with 'A'.
- [...] This is called a character set. It matches any one of the characters that are enclosed in the square brackets. For example:

```
[MVX]
```

matches any one of the characters M, V, or X in a string. Ranges of characters are indicated by using a hyphen between the beginning and ending characters, and enclosing the whole thing in brackets. For example:

```
[0-9]
```

matches any digit. To match '-', write it as '---', which is a range containing only '-'. You may also give '-' as the first or last character in the set. To match '^', put it anywhere except as the first character of a set. To match a ']', make it the first character in the set. For example:

```
[ ]d^]
```

matches either ']', 'd' or '^'.
- [^ ...]

This is a complemented character set. The first character after the [must be a ^. It matches any characters except those in the square brackets (or newline). For example:

```
[^0-9]
```

matches any character that is not a digit.
- |

Vertical bar is the alternation operator and it is used to specify alternatives. For example:

`^P|[0-9]`

matches any string that matches either `^P` or `[0-9]`. This means it matches any string that contains a digit or starts with P. The alternation applies to the largest possible regexps on either side.

No spaces are allowed between strings and the alternation operator.

`{...}`

Brackets are used for grouping in regular expressions as in arithmetic. They can be used to concatenate regular expressions containing the alternation operator, |.

- * Asterisk means that the preceding regular expression is to be repeated as many times as possible to find a match. For example:

`ph*`

applies the `*` symbol to the preceding `h` and looks for matches to one `p` followed by any number of `h`'s. This will also match just `p` if no `h`'s are present. The `*` repeats the smallest possible preceding expression (use parentheses if you wish to repeat a larger expression). It finds as many repetitions as possible. For example:

`(c[ad][ad]*r x)`

matches a string of the form `(car x)`, `(cdr x)`, `(cadr x)`, and so on.

- + Plus sign is similar to `*`, but the preceding expression must be matched at least once. This means that:

`wh+y`

would match `"why"` and `"whhy"` but not `"wy"`, whereas `wh*y` would match all three of these strings. This is a simpler way of writing the last `*` example:

`(c[ad]+r x)`

- ? Question mark is similar to `*`, but the preceding expression can be matched once or not at all. For example:

`fe?d`

will match `fed` and `fd`, but nothing else.

- \ Backslash is used to suppress the special meaning of a character when matching. For example:

`\$`

matches the character `$`.

In regular expressions, the *, +, and ? operators have the highest precedence, followed by concatenation, and finally by |.

Note:

There is now extended support for the MATCH function when used with the FILTER attribute of any VIEW structure that accesses SQL and ODBC back ends. For more detailed information, see the Use of MATCH with PROP:Filter and SQL Databases help topic.

Return Data Type: LONG

Example:

```
A STRING('Richard')
B STRING('RICHARD')
C STRING('R*')
D STRING('[A-D]')
ListHave1     STRING('IN,OH,KY,TN,PA')
ListHave2     STRING('WI,MN,IA,SD,ND')
StatesWanted  STRING('NJ|NY|PA|DE')
RV           BYTE !Return Value

CODE
RV = MATCH(A,B,Match:Simple+Match:NoCase)    !Returns true - case insensitive match
RV = MATCH(A,B,Match:Soundex)              !Returns true - same soundex values
RV = MATCH(A,C)                            !Returns true - wildcard match

RV=MATCH('Fireworks on the fourth', '{{4|four}th', Match:Regular+Match:NoCase)
!returns True
RV=MATCH('July 4th fireworks', '{{4|four}th', Match:Regular+Match:NoCase)
!returns True

IF MATCH(EmployeeName, '^Th?om{[as|my]?{[ ]}+', Match:Regular+Match:NoCase)
  Message('Welcome Tom Thom Thomas or Tommy')
END

!Regular expression for testing an email address as valid.
!1. The name portion can contain the characters: A-Z 0-9 -._
!2. Then must have an @
!3. Then repeating groups containing: A-Z 0-9 -._
!4. A period and 2, 3, 4 letters

RV = MATCH(UPPER(CLIP(eMailAddr)), |
'^[-A-Z0-9._]+@[{-A-Z0-9._}+.[A-Z][A-Z][A-Z]?[A-Z]?$', Match:Regular)
```

See Also: STRPOS

MAXIMUM (return maximum subscript value)

MAXIMUM(*variable,subscript*)

MAXIMUM	Returns maximum subscript value.
<i>variable</i>	The label of a variable declared with a DIM attribute.
<i>subscript</i>	A numeric constant for the subscript number. The <i>subscript</i> identifies which array dimension is passed to the procedure.

The **MAXIMUM** procedure returns the maximum subscript value for an explicitly dimensioned variable. MAXIMUM does not operate on the implicit array dimension of STRING, CSTRING, or PSTRING variables. This is usually used to determine the size of an array passed as a parameter to a procedure or procedure.

Return Data Type: LONG

Example:

```

Array BYTE,DIM(10,12)      !Define a two-dimensional array

!For the above Array:    MAXIMUM(Array,1) returns 10
!                          MAXIMUM(Array,2) returns 12

CODE
LOOP X# = 1 TO MAXIMUM(Array,1)  !Loop until end of 1st dimension
  LOOP Y# = 1 TO MAXIMUM(Array,2) !Loop until end of 2nd dimension
    Array[X#,Y#] = 27           !Initialize each element to default
  END
END

```

See Also:

DIM

Prototype Parameter Lists (Passing Arrays)

MESSAGE (return message box response)

MESSAGE(*text* [,*caption*] [,*icon*] [,*buttons*] [,*default*] [,*style*])

MESSAGE	Displays a message dialog box and returns the button the user pressed.
<i>text</i>	A string constant or variable containing the text to display in the message box. A vertical bar () in the text indicates a line break for multi-line messages. Including '<9>' in the text inserts a tab for text alignment.
<i>caption</i>	The dialog box title. If omitted, the dialog has no title.
<i>icon</i>	A string constant or variable naming the .ICO file to display, or an EQUATE for one of Windows' standard icons (these EQUATES are listed in EQUATES.CLW). If omitted, no icon is displayed on the dialog box.
<i>buttons</i>	Either an integer expression which indicates which Windows standard buttons (may indicate multiple buttons) to place on the dialog box, or a string expression containing a vertical bar () delimited list of the text for up to 8 buttons. If omitted, the dialog displays an Ok button.
<i>default</i>	An integer constant, variable, EQUATE, or expression which indicates the default button on the dialog box. If omitted, the first button is the default.
<i>style</i>	The <i>style</i> parameter is a bitmap integer constant, variable, EQUATE, or expression that specifies the type of modal behavior, and whether or not the text of the message can be copied to the Windows Clipboard.

The actual values for the *style* parameter are:

Dec Bin Type

Modal Type:

0	0b	Application Modal
1	1b	System Modal

Copy Text:

0	00b	message text is displayed as a static text without copy capability
2	10b	message text is displayed as a read only multi-line edit control with the possibility to select all or any part of the text and copy it to the clipboard

The **MESSAGE** procedure displays a Windows-standard message box, typically requiring only a Yes or No response, or no specific response at all. You can specify the font for MESSAGE by setting SYSTEM{PROP:FONT}.

The EQUATES.CLW file contains symbolic constants for the *icon*, *buttons*, and *default* parameters. The following list is all the EQUATEs available for use in the *buttons* and *default* parameters for use when the *buttons* parameter is not a string:

BUTTON:OK
BUTTON:YES
BUTTON:NO
BUTTON:ABORT
BUTTON:RETRY
BUTTON:IGNORE
BUTTON:CANCEL
BUTTON:HELP

When *buttons* is a string, the *default* must be an integer in the range of 1 to the number of buttons defined in the *buttons* text (a maximum of 8).

The MESSAGE procedure returns the number of the button the user presses to exit the dialog box. The button number returned is the constant value that each of these EQUATEs represents (when the *buttons* parameter is an integer), and an integer in the range of 1 to the number of buttons defined in the *buttons* text (up to 8) when *buttons* contains string text.

The following list shows the most common EQUATEs used in the *icon* parameter (there are more listed in EQUATES.CLW):

ICON:None
ICON:Application
ICON:Hand
ICON:Question
ICON:Exclamation
ICON:Asterisk
ICON:Pick
ICON:Clarion

The *style* parameter determines whether the message window is Application Modal or System Modal, and whether or not the message text can be copied to the Windows Clipboard. An Application Modal window must be closed before the user is allowed to do anything else in the application, but does not prevent the user from switching to another Windows application. A System Modal window must be closed before the user is allowed to do anything else in Windows.

The following list shows the EQUATEs used in the *style* parameter:

MSGMODE:SYSMODAL
MSGMODE:CANCOPY

Return Data Type: UNSIGNED

Example:

```
!A ? icon with Yes and No buttons, the default button is No:
CASE MESSAGE('Quit?','Editor',ICON:Question,BUTTON:Yes+BUTTON:No,BUTTON:No,1)
OF BUTTON:No
    !the window is System Modal,no copy
    ability
    CYCLE
OF BUTTON:Yes
    MESSAGE('Goodbye|So Long|Sayonara') !A 3-line message with only an Ok button.
    RETURN
END

!Yes, No, and Maybe buttons, default is Maybe, Application Modal, with copy ability
CASE MESSAGE('Quit?','Editor',ICON:Question,'&Yes|&No|&Maybe',3,2)
OF 1
    !Yes button
    RETURN
OF 2
    !No button
    CYCLE
OF 3
    !Maybe button
    MESSAGE('You have a 50-50 change of staying or going')
    IF CLOCK() % 2
        !Is the current time an odd or even hundredth of a s
        RETURN
    ELSE
        CYCLE
    END
END
```

MONTH (return month of date)

MONTH(*date*)

MONTH Returns month in year.

date A numeric constant, variable, expression, or the label of a STRING, CSTRING, or PSTRING variable declared with a date picture token. The *date* must be a standard date. A variable declared with a date picture token is automatically converted to a standard date intermediate value.

The **MONTH** procedure returns the month of the year (1 to 12) for a given standard date.

Return Data Type: **LONG**

Example:

```
PayMonth = MONTH(DueDate)    !Get the month from the date
```

See Also:

Standard Date

DAY

YEAR

TODAY

DATE

MOUSEX (return mouse horizontal position)

MOUSEX()

The **MOUSEX** procedure returns a numeric value corresponding to the current horizontal position of the mouse cursor at the time of the event. The position is relative to the origin of that window.

The default return value is in dialog units, unless modified by the PROP:Pixels property. If the PROP:Pixels property is set to TRUE (1), the return value of MOUSEX is expressed in pixels.

Return Data Type: SIGNED

Example:

```
SaveMouseX = MOUSEX() !Save mouse position
```

See Also:

MOUSEY

MOUSEY (return mouse vertical position)

MOUSEY()

The **MOUSEY** procedure returns a numeric value corresponding to the current vertical position of the mouse cursor at the time of the event. The position is relative to the origin of that window.

The default return value is in dialog units, unless modified by the PROP:Pixels property. If the PROP:Pixels property is set to TRUE (1), the return value of MOUSEY is expressed in pixels.

Return Data Type: SIGNED

Example:

```
SaveMouseY = MOUSEY()      !Save mouse position
```

See Also:

MOUSEX

NAME (return file name)

NAME(*label*)

NAME Returns name of a file.

label The label of a FILE declaration.

The **NAME** procedure returns a string containing the operating system device name for the structure identified by the *label*. For FILE structures, if the file is OPEN, the complete DOS file specification (drive, path, name, and extension) is returned. If the FILE is closed, the contents of the NAME attribute on the FILE are returned. If there is no NAME attribute, the FILE label is returned.

Return Data Type: **STRING**

Example:

```
OpenFile = NAME(Customer) !Save the name of the open file
```

NEXT (read next record in sequence)

NEXT(*entity*)

NEXT Reads the next record in sequence from a FILE or VIEW.

entity The label of a FILE or VIEW declaration.

NEXT reads the next record in sequence from a FILE or VIEW. The SET (or RESET) statement determines the sequence in which records are read. The first NEXT following a SET reads the record at the position specified by the SET statement. Subsequent NEXT statements read subsequent records in that sequence. The sequence is not affected by any GET, REGET, ADD, PUT, or DELETE. Executing NEXT without a preceding SET, or attempting to read past the end of file posts the "Record Not Available" error.

FILE Usage

NEXT reads the next record in sequence from the data FILE and places it in the RECORD structure data buffer.

VIEW Usage

NEXT reads the next record(s) in sequence from a VIEW and places the appropriate fields in the VIEW structure component files' data buffer(s). If the VIEW contains JOIN structures, NEXT retrieves the appropriate next set of related records.

Either the last SET statement issued on the VIEW's primary file before the OPEN(view) statement, or the SET(view) statement issued after the OPEN(view) determines the sequence in which records are read.

Errors Posted: 33 Record Not Available
 37 File Not Open
 43 Record Is Already Held

Example:

```
ViewOrder VIEW(Customer)                                !Declare VIEW structure
    PROJECT(Cus:AcctNumber,Cus:Name)
    JOIN(Hea:AcctKey,Cus:AcctNumber)                    !Join Header file
    PROJECT(Hea:OrderNumber)
    JOIN(Dtl:OrderKey,Hea:OrderNumber)                  !Join Detail file
    PROJECT(Det:Item,Det:Quantity)
    END
    END
    END
```

```
CODE
OPEN(Customer,22h)
SET(Cus:NameKey)           !Beginning of file in keyed sequence
LOOP                       !Read all records through end of file
  NEXT(Customer)          !read a record sequentially
  IF ERRORCODE() THEN BREAK. !break on end of file
  DO PostTrans            !call transaction posting routine
END

OPEN((Header,22h)
OPEN(Detail,22h)
OPEN(Product,22h)
SET(Cus:AcctKey)
OPEN(ViewOrder)
  LOOP                       !Read all records through end of primary file
  NEXT(ViewOrder)         !read a record sequentially
  IF ERRORCODE() THEN BREAK. !break on end of file
  DO PostTrans            !call transaction posting routine
END                          !End loop
```

See Also:

SET

RESET

PREVIOUS

EOF

HOLD

WATCH

NOMEMO (read file record without reading memo)

NOMEMO(*file*)

NOMEMO Arms "memoless" record retrieval.

file The label of a FILE.

The **NOMEMO** statement arms "memoless" record retrieval for the next GET, REGET, NEXT, or PREVIOUS statement encountered. The following GET, REGET, NEXT, or PREVIOUS gets the record but does not get any associated MEMO or BLOB field(s) for the record. Generally, this speeds up access to the record when the contents of the MEMO or BLOB field(s) are not needed by the procedure.

Example:

```

SET(Master)
LOOP
  NOMEMO(Master)           !Arm "memoless" access
  NEXT(Master)            !Get record without memo
  IF ERRORCODE() THEN BREAK.
  Queue = Mst:Record      !Fill memory queue
  ADD(Queue)
  IF ERRORCODE() THEN STOP(ERROR()).
END
DISPLAY(?ListBox)        !Display the queue

```

See Also:

GET

NEXT

PREVIOUS

MEMO

NOTIFY (send safe information to a receiver thread)

NOTIFY(*notifycode*, *<thread>*, *<parameter>*)

NOTIFY	Sends an event and optional parameter to a receiving thread.
<i>notifycode</i>	An unsigned integer value that indicates the notification or request code.
<i>thread</i>	A signed integer that identifies the number of the receiver thread; if omitted or equal to 0, the current thread is the receiver.
<i>parameter</i>	An optional LONG value that is used as a parameter of the notification or request

The **NOTIFY** statement is called on the sender side. It generates the EVENT:Notify event and places it at the front of the event queue of receiver's thread top window. Generally, the EVENT:Notify event is a special event that can transfer up to 2 additional parameters (*thread* and *parameter*) to the receiver.

Execution of the sender thread continues immediately. It does not wait for any response from the receiver.

NOTIFY and NOTIFICATION are a functional replacement for the SETTARGET(,thread) statement. They can also be used for safe transfer information between threads.

The code between SETTARGET(*,thread*) and a subsequent SETTARGET is executing in the context of the specified *thread* but also as a part of the current thread. Hence, two threads can execute the code in the same context. Also, related DLLs can associate their thread dependent data with the Thread ID of the current thread. These aspects make using SETTARGET(*,thread*) potentially dangerous in the new threading model.

The purpose of NOTIFY and NOTIFICATION is to provide a way to send a notification with parameters from one thread to another. The receiver thread can then execute the request in the correct context.

Example:

```
DynMenu.Construct PROCEDURE()
  CODE
  SELF.NofWindows = 0
  NOTIFY (NOTIFY:Load, 1, ADDRESS (SELF.IDynMenu)) !Send Notify event to primary thread
  RETURN
```

! -----

```
DynMenu.Destruct PROCEDURE()
  CODE
  NOTIFY (NOTIFY:Unload, 1, ADDRESS (SELF.IDynMenu))
  RETURN
```

See Also: NOTIFICATION

NOTIFICATION (receive information from sender thread)

NOTIFICATION(*notifycode*, <*thread*>, <*parameter*>), **BYTE**

NOTIFICATION	Receives information from a sender thread.
<i>notifycode</i>	an UNSIGNED variable that receives a notify code value passed by the sender with a NOTIFY statement.
<i>thread</i>	an optional SIGNED variable that receives the number of the sender's thread parameter.
<i>parameter</i>	a LONG variable that receives the parameter passed by the sender with a NOTIFY statement.

The **NOTIFICATION** function is used by a receiving thread. It receives the notification code, thread number, and parameter passed by the sending thread's NOTIFY statement.

NOTIFICATION returns FALSE (0) if the EVENT() function returns an event other than EVENT:Notify. If EVENT:Notify is posted, **NOTIFICATION** returns TRUE. Because calls to NOTIFY and **NOTIFICATION** are asynchronous, the sender thread can be closed already when receiver thread accepts the EVENT:Notify event.

NOTIFY and NOTIFICATION are a functional replacement for the SETTARGET(,thread) statement. They can also be used for safe transfer information between threads.

The code between SETTARGET(,thread) and a subsequent SETTARGET is executing in the context of the specified *thread* but also as a part of the current thread. Hence, two threads can execute the code in the same context. Also, related DLLs can associate their thread dependent data with the Thread ID of the current thread. These aspects make using SETTARGET(,thread) potentially dangerous in the new threading model.

The purpose of NOTIFY and NOTIFICATION is to provide a way to send a notification with parameters from one thread to another. The receiver thread can then execute the request in the correct context.

Example:

```

CASE EVENT()
  OF EVENT:Accepted
    CASE ACCEPTED()
      OF ?Start
        START (T1)
      OF ?Load
        CALL ('DLL.DLL', 'EntryPoint', 1)
      ELSE
        Q.Feq = ACCEPTED()
        GET (Q, Q.Feq)
        IF ERRORCODE() = 0 AND Q.Op <> 0
          DM &= Q.ID + 0
          DM.ExecuteMenu (Q.Op)
        END
      END
  OF EVENT:Notify
    !NOTIFY has sent a Notify Event. Get Code and Parameter
    IF NOTIFICATION (NCode,, NParam)
      DM &= NParam + 0           !Assign passed parameter to reference var

      CASE Ncode                 !Test the Notify Code
      OF NOTIFY:Load             !Execute appropriate action
        DM.CreateMenu (Q)
      OF NOTIFY:Unload
        DO DestroyMenu
        UNLOAD ('DLL.DLL')      !Execute appropriate action
      END
    END
  END
END

```

See Also: NOTIFY

NULL (return null file field)

NULL(*field*)

NULL	Determines null "value" of a <i>field</i> .
<i>field</i>	The label (including prefix) of a field in a FILE structure. This may be a GROUP or RECORD structure.

The **NULL** procedure returns a non-zero value (true) if the *field* is null, and zero (false) if the *field* contains any known value (including blank or zero). If the *field* is a GROUP or RECORD structure, all component fields of the GROUP or RECORD must be null for NULL to return true. Support for null "values" in a FILE is entirely dependent upon the file driver.

Return Data Type: LONG

Example:

```
Customer FILE,DRIVER('MSSQL'),PRE(Cus) !Declare customer file layout
AcctKey KEY(Cus:AcctNumber)
Record RECORD
AcctNumber LONG
OrderNumber LONG
Name STRING(20)
Addr STRING(20)
CSZ STRING(35)
END
END
```

```
Header FILE,DRIVER('MSSQL'),PRE(Hea) !Declare header file layout
AcctKey KEY(Hea:AcctNumber)
Record RECORD
AcctNumber LONG
OrderNumber LONG
ShipToName STRING(20)
ShipToAddr STRING(20)
ShipToCSZ STRING(35)
END
END
```

```
CODE
OPEN(Header)
OPEN(Customer)
SET(Hea:AcctKey)
LOOP
NEXT(Header)
IF ERRORCODE() THEN BREAK.
IF NULL(Hea:ShipToName) !Check for null ship-to address
Cus:AcctNumber = Hea:AcctNumber
GET(Customer,Cus:AcctKey) !Get Customer record
IF ERRORCODE()
CLEAR(Cus:Record)
END
Hea:ShipToName = Cus:Name ! and assign customer address
Hea:ShipToAddr = Cus:Addr ! as the ship-to address
Hea:ShipToCSZ = Cus:CSZ
END
PUT(Header) !Put Header record back
END
```

See Also:

SETNULL

SETNONNULL

NUMERIC (return numeric string)

NUMERIC(*string*)

NUMERIC Validates all numeric string.

string A string constant, variable, or expression.

The **NUMERIC** procedure returns the value 1 (true) if the *string* only contains a valid numeric value. It returns zero (false) if the *string* contains any non-numeric characters. Valid numeric characters are the digits 0 through 9, a leading minus sign, and a decimal point. **DEFORMAT** is used to return unformatted numbers from a formatted string.

Return Data Type: **UNSIGNED**

Example:

```
!NUMERIC('1234.56')    returns 1
!NUMERIC('1,234.56')   returns 0
!NUMERIC('-1234.56')   returns 1
!NUMERIC('1234.56-')   returns 0
```

```
IF NOT NUMERIC(PartNumber)    !If part number is not numeric
  DO ChkValidPart            !check for valid part number
END                            !End if
```

See Also:

DEFORMAT

OMITTED (return omitted parameters)

OMITTED(*position*)
(*name*)

OMITTED	Tests for unpassed parameters.
<i>position</i>	An integer constant or variable which specifies the ordinal parameter position to test.
<i>name</i>	The name of a procedure parameter.

The **OMITTED** procedure tests whether a parameter of a PROCEDURE was actually passed. The return value is one (true) if the parameter in the specified *position* was omitted. The return value is zero (false) if the parameter was passed. Any *position* past the last parameter passed is considered omitted.

For the purpose of the OMITTED procedure, a parameter may only be omitted if its data type is enclosed in angle brackets (< >) in the PROCEDURE prototype . Although parameters prototyped with default values may be omitted from the procedure call, the default value is actually passed, and the OMITTED procedure therefore returns false (0) for those parameters.

All CLASS methods have an implicit first parameter which is always passed--the CLASS name. This means that OMITTED(1) will always return false for a CLASS method. Any actual parameters passed to the method are numbered beginning with two (2). Therefore, to test whether two actual parameters to a CLASS method are passed means you must test *positions* two (2) and three (3).

The name of a procedure parameter can now be used in the call to the OMITTED() function. The OMITTED function now checks that its parameter is an identifier that matches the name of any procedure parameter it is called from. Otherwise, it treats a passed value as an expression that returns a parameter ordinal number. The identifier passed to OMITTED is taken as is, i.e., if the identifier is an EQUATE that names a label that happens to match the name of some procedure parameter, the compiler does not use the equated label, but uses the EQUATE as the literal procedure parameter.

Return Data Type: LONG

Example:

```

MAP
SomeProc  PROCEDURE (STRING, <LONG>, <STRING>)           !Procedure prototype
END
MyClass  CLASS
MyMethod  PROCEDURE (STRING Field1, <LONG Date>, <STRING Field3>) !Method prototype
END
CODE
SomeProc(Field1,,Field3) !For this statement:
!OMITTED(1) returns 0,OMITTED(2) returns 1,OMITTED(3) returns 0,OMITTED(4)returns 1

```

```

SomeProc PROCEDURE(Field1,Date,Field3)
CODE
IF OMITTED(2)           !If date parameter was omitted
  Date = TODAY()       !substitute the system date
END
MyClass.MyMethod  PROCEDURE(STRING Field1,<LONG Date>,<STRING Field3>)
CODE
IF OMITTED(3)           !If date parameter was omitted
  Date = TODAY()       !substitute the system date
END

*****Example of using OMITTED with a Procedure parameter:
PROGRAM

MAP
  PP(LONG),LONG,TYPE
END

T      CLASS
F      PROCEDURE (<LONG>,<QUEUE>,<FILE>,<KEY>,<BLOB>,<T>, |
          <WINDOW>,<?>,<PP>),LONG,PROC
      END

CODE
T.F()

T.F PROCEDURE (<LONG L>,<QUEUE Q>,<FILE F>,<KEY K>,<BLOB B>,<T TT>, |
          <WINDOW W>,<? A>,<PP P>)

Res  LONG,AUTO

CODE
Res  = 0
Res += OMITTED (SELF)
Res += OMITTED (L)
Res += OMITTED (Q)
Res += OMITTED (F)
Res += OMITTED (K)
Res += OMITTED (B)
Res += OMITTED (TT)
Res += OMITTED (W)
Res += OMITTED (A)
Res += OMITTED (P)
RETURN Res

```

See Also: [Prototype Parameter Lists \(Passing Arrays\)](#)

OPEN (open a data structure)

OPEN(*entity* [, *access mode* / *owner*])

OPEN	Opens a FILE, VIEW, APPLICATION, WINDOW, or REPORT structure for processing.
<i>entity</i>	The label of a FILE, VIEW, APPLICATION, WINDOW, or REPORT structure.
<i>access mode</i>	A numeric constant, variable, or expression which determines the level of access granted to both the user opening the FILE <i>entity</i> , and other users in a multi-user system. If omitted, the default value is 22h (Read/Write + Deny Write). Valid only when the <i>entity</i> parameter names a FILE structure.
<i>owner</i>	The label of the APPLICATION or WINDOW structure which "owns" the window <i>entity</i> being opened. Normally, this parameter would be an &WINDOW reference variable. Valid only when the <i>entity</i> parameter names an APPLICATION or WINDOW structure.

The **OPEN** statement opens a FILE, VIEW, APPLICATION, WINDOW, or REPORT structure for processing.

FILE Usage

The OPEN statement opens a FILE structure for processing and sets the *access mode*. Support for various *access modes* are file driver dependent. All files must be explicitly opened before the records may be accessed.

The *access mode* is a bitmap which tells the operating system what access to grant the user opening the file and what access to deny to others using the file.

The actual values for each access level are:

Dec Hex Access

User Access:

0	0h	Read Only
1	1h	Write Only
2	2h	Read/Write

Other's Access:

0	0h	Any Access (FCB compatibility mode)
16	10h	Deny All
32	20h	Deny Write
48	30h	Deny Read
64	40h	Deny None

VIEW Usage

The OPEN statement opens a VIEW structure for processing. A VIEW must be explicitly opened before it may be accessed. The files used in the VIEW must already be open.

Before the OPEN(*view*) statement, you may issue a SET statement to the VIEW structure's primary file to setup sequential processing for the VIEW. You cannot issue a SET statement to the primary file while the VIEW is OPEN--you must CLOSE(*view*), then issue the SET, and then OPEN(*view*) again. SET(*view*) may be issued while the VIEW is open to setup sequential processing using the ORDER attribute.

Window Usage

OPEN activates an APPLICATION or WINDOW for processing. However, nothing is displayed until a DISPLAY statement or the ACCEPT loop is encountered. This allows an opportunity to execute pre-display code to customize the display.

A *window* with an *owner* always appears on top of its *owner*, and is automatically hidden if the *owner* is minimized or hidden. If the *owner* is closed, all owned windows are also automatically closed. MDI windows are implicitly owned by the frame window. Non-MDI windows do not have an *owner* by default.

REPORT Usage

OPEN activates a REPORT structure. You must explicitly OPEN a REPORT before any of the structures may be printed.

Errors Posted:	02	File Not Found
	03	Path Not Found
	04	Too Many Open Files
	05	Access Denied
	32	File Is Already Locked
	36	Invalid Data File
	38	Invalid Key File
	45	Invalid File Name
	46	Key Files Must Be Rebuilt
	47	Invalid File Declaration
	52	File Already Open
	57	Invalid Memo File
	73	Memo File is Missing
	75	Invalid Field Type Descriptor
	79	Unsupported Data Type In File
	88	Invalid Key Length
	90	File System Error
	92	Build In Progress

Example:

```

ReadOnly    EQUATE(0)                !Access mode equates
WriteOnly   EQUATE(1)
ReadWrite   EQUATE(2)
DenyAll    EQUATE(10h)
DenyWrite  EQUATE(20h)
DenyRead   EQUATE(30h)
DenyNone   EQUATE(40h)

Header      FILE,DRIVER('Clarion'),PRE(Hea)    !Declare header file layout
AcctKey     KEY(Hea:AcctNumber)
OrderKey    KEY(Hea:OrderNumber)
Record      RECORD
AcctNumber  LONG
OrderNumber LONG
ShipToName  STRING(20)
ShipToAddr  STRING(20)
            END
            END

Detail      FILE,DRIVER('Clarion'),PRE(Dt1)     !Declare detail file layout
OrderKey    KEY(Dt1:OrderNumber)
Record      RECORD
OrderNumber LONG
Item        LONG
Quantity    SHORT
            END
            END

ViewOrder   VIEW(Header),ORDER('+Hea:OrderNumber') !Declare VIEW structure
            PROJECT(Hea:OrderNumber)
            JOIN(Dt1:OrderKey,Hea:OrderNumber)    !Join Detail file
            PROJECT(Det:Item,Det:Quantity)
            END
            END

CODE
OPEN(Names,ReadWrite+DenyNone)                !Open fully shared access
OPEN(Header)
OPEN(Detail)
SET(Hea:AcctKey)                               !Set to primary file
OPEN(ViewOrder)                               !then Open view
SET(ViewOrder)                                !or SET(view) after opening
                                                !to use ORDER attribute
OPEN(CustRpt)                                  !Open a report

```

```
Win1Proc PROCEDURE
Win1 WINDOW,ALRT(F10Key)
    END
    CODE
    OPEN(Win1)                !Open the window
    GlobalWindowReference &= Win1    !Assign window reference to a
    !global &WINDOW
    ACCEPT
    IF EVENT() = EVENT:AlertKey
        START(Win2Proc)
    END
    END
    END
```

```
Win2Proc PROCEDURE
Win2 WINDOW
    END
    CODE
    OPEN(Win2,GlobalWindowReference)    !Open Win2, "owned" by Win1
    ACCEPT
    END
```

See Also:

SHARE, CLOSE, SET, FILE, VIEW, APPLICATION, WINDOW, REPORT, ACCEPT

DISPLAY, CLOSE

PACK (remove deleted records)

PACK(*file*)

PACK Removes deleted records from a data file and rebuilds its keys.

file The label of a FILE declaration.

The **PACK** statement removes deleted records from a data file and rebuilds its keys. The resulting data files are as compact as possible. PACK requires at least twice the disk space that the file, keys, and memos occupy to perform the process. New files are created from the old, and the old files are deleted only after the process is complete. PACK requires exclusive access to the file. Therefore, the file must be opened with *access mode* set to 12h (Read/Write Deny All) or 22h (Read/Write Deny Write).

PACK will generate events to the currently open window if you assign a value (an integer from 1 to 100) to PROP:ProgressEvents for the affected FILE before you issue the PACK. The larger the value you assign to PROP:ProgressEvents, the more events are generated and the slower the PACK will progress. These events allow you to indicate to the user the progress of the PACK. This can keep end-users informed that PACK is still working while building large files (so they don't re-boot thinking the machine has locked up).

It is not valid to make any calls to the *file* being built except to query its properties, call NAME(*file*), or CLOSE(*file*) (which aborts the process and is not recommended). Issuing a CYCLE statement in response to any of the events generated (except EVENT:BuildDone) cancels the operation. During the PACK operation, *file*{PROP:Completed} returns the percentage completed of the re-build and you can use *file*{PROP:CurrentKey} to get a key reference then either *key*{PROP:Name} or *key*{PROP:Label} to return the name of the current key being built.

Errors Posted: 63 Exclusive Access Required

Events Generated:

EVENT:BuildFile PACK(*file*) is rebuilding the data portion of the *file*.

EVENT:BuildKey PACK(*file*) is rebuilding the keys in the *file*.

EVENT:BuildDone The PACK is complete.

Example:

```
OPEN(Trans,12h)        !Open the file in exclusive mode
PACK(Trans)            ! and pack it
```

See Also: OPEN, SHARE, BUILD. PROP:ProgressEvents, PROP:Completed

PATH (return current directory)

PATH()

PATH returns a string containing the current drive and directory. This is equivalent to the **SHORTPATH** procedure.

PATH, can be used in expressions evaluated by the **EVALUATE** statements in runtime applications and in the templates.

Return Data Type: **STRING**

Example:

```
IF PATH() = 'C:\'           !If in the root
  MESSAGE('You are in the Root Directory') !display message
END
```

See Also:

SETPATH

SHORTPATH

LONGPATH

DIRECTORY

PEEK (read memory address)

PEEK(*address,destination*)

PEEK	Reads data from a memory address.
<i>address</i>	A numeric constant, variable, or expression (evaluated to a LONG) that specifies a memory address. This parameter should always use the ADDRESS procedure, to ensure the correct address is used.
<i>destination</i>	The label of a variable to receive the contents of the memory location.

The **PEEK** statement reads the memory address specified by *address* and copies the data found there into the *destination* variable. PEEK reads as many bytes as are required to fill the *destination* variable.

It is easily possible to create a General Protection Fault (GPF) if you PEEK at an incorrect address, so great care should be taken when using PEEK. There are usually Windows API procedures that will do whatever you require of PEEK and these should be used in preference to PEEK.

Example:

```
MemAddress      LONG
Dest1           BYTE
Dest2           SHORT
Dest3           REAL
KeyboardFlag    BYTE
```

```
CODE
  PEEK(ADDRESS(MemAddress),Dest1)      !Read 1 byte
  PEEK(ADDRESS(MemAddress),Dest2)      !Read 2 bytes
  PEEK(ADDRESS(MemAddress),Dest3)      !Read 8 bytes
  PEEK(ADDRESS(0040h,0017h),KeyboardFlag) !Read keyboard status byte
```

See Also:

POKE

ADDRESS

PENCOLOR (return line draw color)

PENCOLOR()

The **PENCOLOR** procedure returns the current pen color set by **SETPENCOLOR**.

Return Data Type: LONG

Example:

```

Proc1    PROCEDURE
MDIChild1 WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
        !window controls
        END

CODE
OPEN(MDIChild1)
SETPENCOLOR(000000FFh)      !Set blue pen color
Proc2                                !Call another procedure

Proc2    PROCEDURE
MDIChild2 WINDOW('Child Two'),AT(0,0,320,200),MDI,MAX,HVSCROLL
        !window controls
        END

ColorNow    LONG
CODE
ColorNow = PENCOLOR()      !Get current pen color
OPEN(MDIChild2)
SETPENCOLOR(ColorNow)     !Set same pen color
SETPENSTYLE(PEN:dash)     !Set dashes for line style
SETPENWIDTH(2)           !Set two dialog unit thickness
BOX(100,50,100,50,00FF0000h) !Red box with thick blue dashed border

```

See Also:

SETPENCOLOR

PENSTYLE (return line draw style)

PENSTYLE()

The **PENSTYLE** procedure returns the current line draw style set by SETPENSTYLE.

EQUATE statements for the pen styles are contained in the EQUATES.CLW file. The following list is a representative sample of these (see EQUATES.CLW for the complete list):

PEN:solid	Solid line
PEN:dash	Dashed line
PEN:dot	Dotted line
PEN:dashdot	Mixed dashes and dots

Return Data Type: SIGNED

Example:

```

Proc1 PROCEDURE
MDIChild1 WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
    !window controls
    END
    CODE
    OPEN(MDIChild1)
    SETPENCOLOR(00000FFh) !Set blue pen color
    SETPENSTYLE(PEN:dash) !Set dashes for line style
    Proc2 !Call another procedure

Proc2 PROCEDURE
MDIChild2 WINDOW('Child Two'),AT(0,0,320,200),MDI,MAX,HVSCROLL
    !window controls
    END
ColorNow LONG
StyleNow LONG
    CODE
    ColorNow = PENCOLOR() !Get current pen color
    StyleNow = PENSTYLE() !Get current pen style
    OPEN(MDIChild2)
    SETPENCOLOR(ColorNow) !Set same pen color
    SETPENSTYLE(StyleNow) !Set same pen style
    SETPENWIDTH(2) !Set two dialog unit thickness
    BOX(100,50,100,50,00FF0000h) !Red box with thick blue dashed border
  
```

See Also:

SETPENSTYLE

PENWIDTH (return line draw thickness)

PENWIDTH()

The **PENWIDTH** procedure returns the current line draw thickness set by SETPENWIDTH. The return value is in dialog units (unless overridden by the THOUS, MM, or POINTS attributes on a REPORT).

Return Data Type: SIGNED

Example:

```

Proc1  PROCEDURE
MDIChild1 WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
        !window controls
        END

CODE
OPEN(MDIChild1)
SETPENCOLOR(000000FFh)           !Set blue pen color
SETPENSTYLE(PEN:dash)           !Set dashes for line style
SETPENWIDTH(2)                   !Set two dialog unit thickness
Proc2                             !Call another procedure

Proc2  PROCEDURE
MDIChild2 WINDOW('Child Two'),AT(0,0,320,200),MDI,MAX,HVSCROLL
        !window controls
        END

ColorNow LONG
StyleNow LONG
WidthNow LONG
CODE
ColorNow = PENCOLOR()           !Get current pen color
StyleNow = PENSTYLE()           !Get current pen style
WidthNow = PENWIDTH()           !Get current pen width
OPEN(MDIChild2)
SETPENCOLOR(ColorNow)           !Set same pen color
SETPENSTYLE(StyleNow)           !Set same pen style
SETPENWIDTH(WidthNow)           !Set same pen width
BOX(100,50,100,50,00FF0000h)    !Red box with thick blue dashed border

```

See Also:

SETPENWIDTH

PIE (draw a pie chart)

PIE(*x* , *y* , *width* , *height* , *slices* , *colors* [, *depth*] [, *wholevalue*] [, *startangle*] [, *attributelist*])

PIE	Draws a pie chart on the current window or report.
<i>x</i>	An integer expression that specifies the horizontal position of the starting point.
<i>y</i>	An integer expression that specifies the vertical position of the starting point.
<i>width</i>	An integer expression that specifies the width.
<i>height</i>	An integer expression that specifies the height.
<i>slices</i>	A SIGNED array of values that specify the relative size of each slice of the pie.
<i>colors</i>	A LONG array that specifies the fill color for each slice.
<i>depth</i>	An integer expression that specifies the depth of the three-dimensional pie chart. If omitted, the chart is two-dimensional.
<i>wholevalue</i>	A numeric constant or variable that specifies the total value required to create a complete pie chart. If omitted, the sum of the <i>slices</i> array is used.
<i>startangle</i>	A numeric constant or variable that specifies the starting point of the first slice of the pie, measured as a fraction of the <i>wholevalue</i> . If omitted (or zero), the first slice starts at the twelve o'clock position.
<i>attributelist</i>	A string constant, variable, or EQUATE containing an optional type of output document and its associated attributes. Only valid when the target is a REPORT. (See <i>EXTEND</i>)

The **PIE** procedure creates a pie chart on the current window or report. The pie (an ellipse) is drawn inside a "bounding box" defined by the *x*, *y*, *width*, and *height* parameters. The *x* and *y* parameters specify the starting point, and the *width* and *height* parameters specify the horizontal and vertical size of the "bounding box."

The slices of the pie are created clockwise from the *startangle* parameter as a fraction of the *wholevalue*. Supplying a *wholevalue* parameter that is greater than the sum of all the *slices* array elements creates a pie chart with a piece missing.

The color of the lines is the current pen color set by **SETPENCOLOR**; the default color is the Windows system color for window text. The width of the lines is the current width set by **SETPENWIDTH**; the default width is one pixel. The line style is the current pen style set by **SETPENSTYLE**; the default style is a solid line.

Example:

```
MDIChild WINDOW( 'Child One' ), AT( 0 , 0 , 320 , 200 ) , MDI , MAX , HVSCROLL
    !window controls
END
```

```
SliceSize  SIGNED,DIM(4)
SliceColor LONG,DIM(4)
```

```
CODE
```

```
SliceSize[1] = 90
SliceColor[1] = 0           !Black
SliceSize[2] = 90
SliceColor[2] = 00FF0000h  !Red
SliceSize[3] = 90
SliceColor[3] = 0000FF00h  !Green
SliceSize[4] = 90
SliceColor[4] = 000000FFh  !Blue
OPEN(MDICHild)

PIE(100,50,100,50,SliceSize,SliceColor)
!Draw pie chart containing
!four equal slices, starting at 12 o'clock
!drawn counter-clockwise - Black, Red, Green, and Blue
```

See Also:

Current Target

SETPENCOLOR

SETPENWIDTH

SETPENSTYLE

POINTER (return last queue entry position)

POINTER(*queue*)

POINTER Returns the entry number of the last entry accessed in a QUEUE.

queue The label of a QUEUE structure, or the label of a passed QUEUE parameter.

The **POINTER** procedure returns a LONG integer specifying the entry number of the last QUEUE entry accessed by ADD, GET, or PUT.

The value of **POINTER** is only valid if the ADD, GET, or PUT operation is successful.

Return Data Type: LONG

Example:

```
Que:Name = 'Jones'           !Initialize key field in sorted queue
GET(NameQue,Que:Name)       !Get the entry, if available
IF ERRORCODE()              !Check for errors
  ADD(NameQue,POINTER(NameQue)) !and add the sorted entry, if missing
END
```

See Also:

GET

PUT

ADD

POKE (write to memory address)

POKE(*address,source*)

POKE	Writes data to a memory address.
<i>address</i>	A numeric constant, variable, or expression (evaluated to a LONG) which specifies a memory address. This parameter should always use the ADDRESS procedure, to ensure the correct protected mode address is used.
<i>source</i>	The label of a variable.

The **POKE** statement writes the contents of the *source* variable to the memory address at *address*. POKE writes as many bytes as are in the *source* variable.

It is easily possible to create a General Protection Fault (GPF) if you POKE to an incorrect address, so great care should be taken when using POKE. There are usually Windows API functions that will do whatever you require of POKE and these should be used in preference to POKE.

Example:

```
MAddress      LONG
Source1       BYTE
Source2       SHORT
Source3       REAL
KeyboardFlag  BYTE
```

```
CODE
POKE(ADDRESS(MAddress),Source1) !Write 1 byte to the memory location

POKE(ADDRESS(MAddress),Source2) !Write 2 bytes to the memory location

POKE(ADDRESS(MAddress),Source3) !Write 8 bytes to the memory location

PEEK(ADDRESS(0040h,0017h),KeyboardFlag)!Read keyboard status byte
KeyboardFlag = BOR(KeyboardFlag,40h) !turn on caps lock
POKE(ADDRESS(0040h,0017h),KeyboardFlag)!and put it back
```

See Also:

PEEK

ADDRESS

POLYGON (draw a multi-sided figure)

POLYGON(*array* [, *fill*] [, *attributelist*])

POLYGON	Draws a multi-sided figure on the current window or report.
<i>array</i>	An array of SIGNED integers that specify the x and y coordinates of each "corner point" of the polygon.
<i>fill</i>	A LONG integer constant, constant EQUATE, or variable containing the red, green, and blue components that create the color in the three low-order bytes (bytes 0, 1, and 2) or an EQUATE for a standard Windows color value.
<i>attributelist</i>	A string constant, variable, or EQUATE containing an optional type of output document and its associated attributes. Only valid when the target is a REPORT. (See <i>EXTEND</i>)

The **POLYGON** procedure places a multi-sided figure on the current window or report. The polygon is always closed.

The *array* parameter contains the x and y coordinates of each "corner point" of the polygon. The polygon will have as many corner points as the total number of array elements divided by two. For each corner point in turn, its x coordinate is taken from the odd-numbered array element and the y coordinate from the immediately following even-numbered element.

The border color is the current pen color set by **SETPENCOLOR**; the default color is the Windows system color for window text. The border width is the current width set by **SETPENWIDTH**; the default width is one pixel. The line's style is the current pen style set by **SETPENSTYLE**; the default style is a solid line.

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
    !window controls
    END
Corners SIGNED,DIM(8)
CODE
Corners[1] = 0           !1st x position
Corners[2] = 90         !1st y position
Corners[3] = 90         !2nd x position
Corners[4] = 190        !2nd y position
Corners[5] = 100        !3rd x position
Corners[6] = 200        !3rd y position
Corners[7] = 50         !4th x position
Corners[8] = 60         !4th y position
OPEN(MDIChild)
POLYGON(Corners,000000FFh) !Blue filled four-sided polygon
```

See Also: Current Target, **SETPENCOLOR**, **SETPENWIDTH**, **SETPENSTYLE**

POPUP (return popup menu selection)

POPUP(*selections* [, *x*] [, *y*])

POPUP	Returns an integer indicating the user's choice from the menu.
<i>selections</i>	A string constant, variable, or expression containing the text for the menu choices.
<i>x</i>	An integer constant, variable, or expression that specifies the horizontal position of the top left corner. If omitted, the menu appear at the current cursor position.
<i>y</i>	An integer constant, variable, or expression that specifies the vertical position of the top left corner. If omitted, the menu appear at the current cursor position.

The **POPUP** procedure returns an integer indicating the user's choice from the popup menu that appears when the procedure is invoked. If the user CLICKS outside the menu or presses ESC (indicating no choice), POPUP returns zero.

Within the *selections* string, each choice in the popup menu must be delimited by a vertical bar (|) character. The following rules apply:

- A set of vertical bars containing only a hyphen (|-|) defines a separator between groups of menu choices.
- A menu choice immediately preceded by a tilde (~) is disabled (it appears dimmed out in the popup menu).
- A menu choice immediately preceded by a plus sign (+) appears with a check mark to its left in the popup menu.
- A menu choice immediately preceded by a minus sign (-) appears without a check mark to its left in the popup menu.
- A menu choice immediately followed by a set of choices contained within curly braces (|SubMenu{{SubChoice 1|SubChoice 2}}|) defines a sub-menu within the popup menu (the two beginning curly braces are required by the compiler to differentiate your sub-menu from a string repeat count).
- You may use the ASCII tab character (<9>) in your *selection* string to right-align text.
- You may specify an icon for the menu item by preceding the menu choice with square brackets enclosing PROP:Icon & '(MyIco.ICO)]MenuChoice'

```
POPUP('[' & PROP:Icon & '(MyIco.ICO)]MenuChoice')
```

Each menu selection is numbered in ascending sequence according to its position within the *selections* string, beginning with one (1). Separators and selections that call a sub-menu are not included in the numbering sequence (which makes an EXECUTE structure the most efficient code structure to use with this procedure). When the user CLICKS or presses ENTER on a choice, the procedure terminates, returning the position number of the selected menu item.

Return Data Type: SIGNED

Example:

```

PopupString = 'First|+Second|Sub menu{{One|Two}}|-|Third|~Disabled|' & |
              '[' & PROP:Icon & '(MyIco.ICO)]Last Menu Choice'
ToggleChecked = 1
ACCEPT
CASE EVENT()
OF EVENT:AlertKey
  IF KEYCODE() = MouseRight
    EXECUTE POPUP(PopupString)
    FirstProc          !Call proc for selection 1
    BEGIN              !Code to execute for toggle selection 2
      IF ToggleChecked = 1 !Check toggle state
        SecondProc(Off)   !Call proc to turn off something
        PopupString[7] = '-' !Reset string so the check mark does not appear
        ToggleChecked = 0 !Set toggle flag
      ELSE
        SecondProc(On)    !Call proc to turn off something
        PopupString[7] = '+' !Reset string so the check mark does appear
        ToggleChecked = 1 !Set toggle flag
      END
    END                !End Code to execute for toggle selection 2
    OneProc            !Call proc for selection 3
    TwoProc            !Call proc for selection 4
    ThirdProc          !Call proc for selection 5
    DisabledProc       !Selection 6 is dimmed so it cannot run this proc
    IconProc           !Selection 7 displays an icon in the menu
  END
END
END
END
END

```

POSITION (return record sequence position)

POSITION(*sequence*)

POSITION Identifies a record's unique position in a FILE or VIEW or QUEUE.

sequence The label of a VIEW, FILE, KEY, or INDEX or QUEUE declaration.

POSITION returns a STRING which identifies a record's unique position within the *sequence*. **POSITION** returns the position of the last record accessed in the file or VIEW. The **POSITION** procedure is used with **RESET** to temporarily suspend and resume sequential processing.

FILE usage

The value contained in the returned STRING and the length of that STRING are dependent on the file driver. As a general rule, for file systems that have record numbers, the size of the STRING returned by **POSITION(file)** is 4 bytes. The return string from **POSITION(key)** is 4 bytes plus the sum of the sizes of the fields in the key. For file systems that do not have record numbers, the size of the STRING returned by **POSITION(file)** is generally the sum of the sizes of the fields in the Primary Key (the first KEY on the FILE that does not have the DUP or OPT attribute). The return string from **POSITION(key)** is the sum of the sizes of the fields in the Primary Key plus the sum of the sizes of the fields in the key.

VIEW usage

The return string for **POSITION(view)** contains all the information required by the underlying file system to reset to the one specific position within the record set currently in the VIEW. It also contains the file system's **POSITION** return value for the primary file key and all secondary file linking keys. This allows **POSITION(view)** to accurately define a position for all related records in the VIEW.

QUEUE usage

POSITION(queue) returns a pointer to the first queue record with a matching key value (for current active sort order). If an exact match is not found, a pointer to the next entry greater to one given in the current queue buffer is returned. If all entries in the queue have a lower key, **RECORDS(queue)+1** is returned.

Errors Posted: 35 Record Not Found

Return Data Type: LONG for **POSITION(QUEUE)**
 STRING all others

Example of (POSITION(VIEW)):

```

ViewO VIEW(Customer) !Declare VIEW structure
  PROJECT(Cus:AcctNumber,Cus:Name)
  JOIN(Hea:AcctKey,Cus:AcctNumber) !Join Header file
  PROJECT(Hea:OrderNumber)
  JOIN(Dtl:OrderKey,Hea:OrderNumber) !Join Detail file
  PROJECT(Det:Item,Det:Quantity)
  JOIN(Pro:ItemKey,Dtl:Item) !Join Product file
  PROJECT(Pro:Description,Pro:Price)
  END
  END
  END
  END
RecordQue QUEUE,PRE(Que)
AcctNumber LIKE(Cus:AcctNumber)
Name LIKE(Cus:Name)
OrderNumber LIKE(Hea:OrderNumber)
Item LIKE(Det:Item)
Quantity LIKE(Det:Quantity)
Description LIKE(Pro:Description)
Price LIKE(Pro:Price)
  END
SavPosition STRING(260)
CODE
OPEN(Customer,22h)
OPEN(Header,22h)
OPEN(Detail,22h)
OPEN(Product,22h)
SET(Cus:AcctKey)
OPEN(ViewOrder) !Top of file in keyed sequence
LOOP !Read all records in file
  NEXT(ViewOrder) !read a record sequentially
  IF ERRORCODE()
    DO DisplayQue !Display the queue
    BREAK
  END
  RecordQue ::= Cus:Record !Move record into queue
  RecordQue ::= Hea:Record !Move record into queue
  RecordQue ::= Dtl:Record !Move record into queue
  RecordQue ::= Pro:Record !Move record into queue
  ADD(RecordQue) !and add it
  ASSERT(~ERRORCODE())
  IF RECORDS(RecordQue) = 20 !20 records in queue?
    DO DisplayQue !Display the queue
  END
END
END

```

```

DisplayQue ROUTINE
  SavPosition = POSITION(ViewOrder)!Save record position
  DO ProcessQue           !Display the queue
  FREE(RecordQue)       !and free it
  RESET(ViewOrder,SavPosition) !Reset the record pointer
  NEXT(ViewOrder)       !and get the record again

```

Example (POSITION(Queue)):

```

TaxQ  QUEUE
LowPay  DECIMAL(9,2)
HighPay DECIMAL(9,2)
TaxAmount DECIMAL(9,2)
PlusPercent DECIMAL(5,2)
      END

Bracket  LONG,AUTO

CODE
SORT(TaxQ, TaxQ.HighPay)           !Set sort order
TaxQ.HighPay = PayCheck.GrossPay !Initialize QUEUE key field
Bracket = POSITION(TaxQ)             !Find pay bracket
IF Bracket > RECORDS(TaxQ)         !If value exceeds number of records
  Bracket = RECORDS(TaxQ)           !in QUEUE, set to highest value in QUEUE
END
GET(TaxQ,Bracket)                  !read QUEUE entry
Paycheck.Tax = TaxAmount+ |         !calculation based on QUEUE entry
(Paycheck.Gross-TaxQ.LowPay)*TaxQ.PlusPercent

```

See Also:

RESET

REGET

POST (post user-defined event)

POST(*event* [,*control*] [,*thread*] [,*position*])

POST	Posts an event.
<i>event</i>	An integer constant, variable, expression, or EQUATE containing an event number. A value in the range 400h to 0FFFh is a User-defined event.
<i>control</i>	An integer constant, EQUATE, variable, or expression containing the field number of the control affected by the event. If omitted, the event is field-independent.
<i>thread</i>	An integer constant, EQUATE, variable, or expression containing the execution thread number whose ACCEPT loop is to process the event. If omitted, the event is posted to the current thread.
<i>position</i>	An integer constant, EQUATE, variable, or expression containing either zero (0) or one (1). If one (1), the <i>event</i> message is placed at the front of the event message queue. If omitted or zero (0), the <i>event</i> message is placed at the end of the event message queue.

POST posts an event to the currently active ACCEPT loop of the specified *thread*. This may be a User-defined event, or any other event. User-defined event numbers can be defined as any integer between 400h and 0FFFh. Any *event* posted with a *control* specified is a field-specific event, while those without are field-independent events.

POSTING an event causes the ACCEPT loop to fire but does not cause the event to "happen." For example, POST(EVENT:Selected,?MyControl) executes any code in EVENT:Selected for ?MyControl but does not cause ?MyControl to gain focus.

Example:

```
Win1  WINDOW( 'Tools' ),AT(156,46,32,28),TOOLBOX
      BUTTON( 'Date' ),AT(0,0,,),USE(?Button1)
      BUTTON( 'Time' ),AT(0,14,,),USE(?Button2)
      END
CODE
OPEN(Win1)
ACCEPT
  IF EVENT() = EVENT:User           !Detect user-defined event
    BREAK
  END
  CASE ACCEPTED( )
  OF ?Button1
    POST(EVENT:User,,UseToolsThread) !Post field-independent event to other thread
  OF ?Button2
    POST(EVENT:User)                 !Post field-independent event to this thread
  END
END;CLOSE(Win1)
```

See Also: ACCEPT, EVENT

PRAGMA (control pragma settings from source)

PRAGMA(*string*)

PRAGMA Send to the project system a statement from your generated source code

string A string constant, variable, or expression, of the format '*project(string)*', '*compile(string)*', or '*link(string)*'.

PRAGMA('project(string)')

Supports the use of the project system's statements in Clarion source code, and adds the statement to the project dynamically. The string parameter defines the proper project syntax.

PRAGMA('compile(string)')

Equivalent to PRAGMA('project(#compile string)')

PRAGMA('link(string)')

Equivalent to PRAGMA('project(#pragma link(string))')

PRAGMA supports the use of the project system's pragma statement in Clarion source code, and adds the pragma to the project dynamically. The *string* parameter defines the proper project syntax.

Examples:

```

PROGRAM
MAP
  INCLUDE( 'MYUTIL.INC' )
END
PRAGMA( 'project(#compile MYUTIL.CLW)' )
PRAGMA( 'project(#pragma link(C%V%DOS%X%L%.LIB))' )

F FILE,DRIVER( 'DOS' ),CREATE,NAME( 'Test.!' )
Record RECORD
  BYTE,DIM(1000)
END

END

CODE
CREATE (F)

```

In the above example, the project **pragma** forces the system to compile MYUTIL.CLW, and automatically adds the correct DOS driver library to the link list.

```
PROGRAM
MAP
  INCLUDE( 'MYUTIL.INC' )
END
PRAGMA( 'compile(MYUTIL.CLW)' )
PRAGMA( 'link(C%V%DOS%X%%L%.LIB)' )

F      FILE,DRIVER( 'DOS' ),CREATE,NAME( 'Test.!' )
Record RECORD
  BYTE,DIM(1000)
  END
  END

CODE
CREATE(F)
```

The above example is identical in function to the first example.

PRESS (put characters in the buffer)

PRESS(*string*)

PRESS Places characters in the keyboard input buffer.

string A string constant, variable, or expression.

PRESS places characters in the Windows keyboard input buffer. The entire *string* is placed in the buffer. Once placed in the keyboard buffer, the *string* is processed just as if the user had typed in the data.

Example:

```
IF LocalRequest = AddRecord      !On the way into a memo on adding a record
  TempString = FORMAT(TODAY(),@D1) & ' ' & FORMAT(CLOCK(),@T4)
  PRESS(TempString)              !Pre-load first line of memo with date and time
END
```

See Also:

PRESSKEY

PRESSKEY (put a keystroke in the buffer)

PRESSKEY(*keycode*)

PRESSKEY Places one keystroke in the keyboard input buffer.

keycode An integer constant or keycode EQUATE label.

PRESSKEY places one keystroke in the Windows keyboard input buffer. Once placed in the keyboard buffer, the *keycode* is processed just as if the user had pressed the key. ALIAS does not transform a **PRESSKEY** *keycode*.

Example:

```
IF Action = 'Add'      !On the way into a memo control on an add record
  Cus:MemoControl = FORMAT(TODAY(),@D1) & ' ' & FORMAT(CLOCK(),@T4)
                      !Pre-load first line of memo with date and time
  PRESSKEY(EnterKey)  !and position user on second line
END
```

See Also:

PRESS

PREVIOUS (read previous view record in sequence)

PREVIOUS(*entity*)

PREVIOUS Reads the previous record in sequence.

entity The label of a FILE or VIEW declaration.

PREVIOUS reads the previous record(s) in sequence from a FILE or VIEW. The SET (or RESET) statement determines the sequence in which records are read. Executing PREVIOUS without a preceding SET, or attempting to read past the beginning of a file posts the "Record Not Available" error.

FILE Usage

PREVIOUS reads the previous record in sequence from a data file and places it in the RECORD structure data buffer. The first PREVIOUS following a SET reads the record at the position specified by the SET statement. Subsequent PREVIOUS statements read subsequent records in reverse sequence. The sequence is not affected by any GET, REGET, ADD, PUT, or DELETE.

VIEW Usage

PREVIOUS reads the previous record(s) in sequence from a VIEW and places the appropriate fields in the VIEW structure component files' data buffer(s). If the VIEW contains JOIN structures, PREVIOUS retrieves the appropriate previous set of related records.

Either the SET statement issued on the VIEW's primary file before the OPEN(*view*) statement, or the SET(*view*) statement issued after the OPEN(*view*) determines the sequence in which records are read. The first PREVIOUS(*view*) reads the record at the position specified by the SET statement. Subsequent PREVIOUS statements read subsequent records in that sequence. The sequence is not affected by PUT or DELETE statements.

Errors Posted: 33 Record Not Available
 37 File Not Open
 43 Record Is Already Held

Example:

```
ViewOrder VIEW(Header)
    PROJECT(Hea:OrderNumber)
    JOIN(Dtl:OrderKey,Hea:OrderNumber) !Join Detail file
    PROJECT(Det:Item,Det:Quantity)
    JOIN(Pro:ItemKey,Dtl:Item) !Join Product file
    PROJECT(Pro:Description,Pro:Price)
    END
    END
    END
CODE
OPEN((Header,22h)
OPEN(Detail,22h)
OPEN(Product,22h)
SET(Cus:AcctKey)
OPEN(ViewOrder)
    LOOP !Read all records through beginning of primary file
    PREVIOUS(ViewOrder) !read a record sequentially
    IF ERRORCODE() THEN BREAK END !break on end of file
    DO PostTrans !call transaction posting routine
    END !End loop
```

See Also:

SET

RESET

NEXT

BOF

HOLD

WATCH

REGET

ADD

PUT

DELETE

PRINT (print a report structure)

```
PRINT( | structure      | )
      | report ,number |
```

PRINT	Prints a report DETAIL structure.
<i>structure</i>	The label of a DETAIL structure.
<i>report</i>	The label of a REPORT structure.
<i>number</i>	The number or EQUATE label of a report DETAIL structure to print (only valid with a <i>report</i> parameter).

The **PRINT** statement prints a report structure to the Windows default printer or the destination specified by the user in the Windows Print... dialog. PRINT automatically activates group breaks and page overflow as needed.

Example:

```
MEMBER( )

MAP
  BuildRpt
  PrintRpt(REPORT, LONG)
END

BuildRpt PROCEDURE
CustRpt  REPORT
        HEADER, USE(?PageHeader)    !Page header
        !structure elements
        END
CustDetail DETAIL, USE(?Detail)      !Line item detail
        !structure elements
        END      !
        END

CODE
PRINT(CustDetail)                    !Print order detail line
PrintRpt(CustRpt, ?PageHeader)       !Pass report and equate to print proc

PrintRpt PROCEDURE(RptToPrint, DetailNumber)
CODE
PRINT(RptToPrint, DetailNumber)     !Print its structure
```

See Also: Page Overflow, BREAK, DETAIL

PRINTERDIALOG (return chosen printer)

PRINTERDIALOG([*title*] [,*flag*])

PRINTERDIALOG

Displays the Windows standard printer choice dialog box to allow the user to choose a file.

title A string constant or variable containing the title to place on the file choice dialog. If omitted, a default *title* is supplied by Windows.

flag A numeric constant or variable which displays a target Print dialog based on the flag's value:
 0 = *Choose Printer* dialog
 1 = *Print Setup* dialog
 2 = *Page Setup* dialog

The **PRINTERDIALOG** procedure displays the Windows standard printer choice dialog box, Print Setup dialog, or Page Setup dialog based on the *flag* value and returns the printer chosen by the user in the PRINTER "built-in" variable in the internal library. This sets the default printer used for the next REPORT opened.

PRINTERDIALOG returns zero (0) if the user pressed the Cancel button, or one (1) if the user pressed the Ok button on the dialog.

Return Data Type: SIGNED

Example:

```
CustRpt REPORT,AT(1000,1000,6500,9000),THOUS,FONT('Arial',12),PRE(Rpt)
      !Report structures and controls
      END
```

```
CODE
IF NOT PRINTERDIALOG('Choose Printer')
  RETURN                    !Abort if user pressed Cancel
END
OPEN(CustRpt)
```

POPERRORS (return error information)

POPERRORS()

POPERRORS Returns the last error information saved by **PUSHERRORS()**

The **POPERRORS** procedure restores the last error information that was saved by **PUSHERRORS**. This includes **ERRORCODE()**, **ERROR()**, **FILEERRORCODE()**, **FILEERROR()**, **ERRORFILE()**, the C library *errno* variable, and the current Windows error code returned by the **GetLastError()** API function. To read these errors, you need to use the appropriate support statements.

POPERRORS is especially useful for programmer's that use the **GetLastError()** API function. **POPERRORS** restores the Windows error code that had been saved by **PUSHERRORS**. **GetLastError()** will return the same error code after **POPERRORS** that it would have returned just prior to **PUSHERRORS**.

PUSHERRORS and **POPERRORS** works using the LIFO (last in, first out) stack convention. The maximum depth of the stack is dependent on available memory.

Because **ERRORCODE()**, **ERROR()**, etc. are thread dependent, **PUSHERRORS** and **POPERRORS** statements that are called from different threads are independent of each other, and allocate a different memory stack for each thread.

Example:

```

PROGRAM

MAP
  InitialErrors()
END

TESTFILE  FILE,DRIVER('DOS')
          RECORD
f1        BYTE
          END
          END

CODE
!file processing and initialization here
OPEN(testfile)  !forcing an error
PUSHERRORS()
!additional code here
InitialErrors  !Call procedure

```

```
InitialErrors      PROCEDURE

ERRORCODEVAR      LONG
ERRORVAR          CSTRING(255)
FILEERRORCODEVAR  LONG
FILEERRORVAR      CSTRING(255)
ERRORFILEVAR      CSTRING(255)
```

```
Window WINDOW('Error Report'),AT(,,362,128),FONT('MS Sans
Serif',8,,FONT:regular,CHARSET:ANSI),SYSTEM, |
    GRAY
    PROMPT('ERRORCODE:'),AT(9,10),USE(?Prompt1)
    STRING(@N_4),AT(97,10,43,10),USE(ERRORCODEVAR)
    PROMPT('ERROR:'),AT(9,25),USE(?Prompt2)
    STRING(@s40),AT(96,25),USE(ERRORVAR)
    PROMPT('FILEERRORCODE:'),AT(9,40),USE(?Prompt3)
    STRING(@N_4),AT(95,40),USE(FILEERRORCODEVAR)
    PROMPT('FILEERROR:'),AT(9,55),USE(?Prompt4)
    STRING(@s40),AT(95,55),USE(FILEERRORVAR)
    PROMPT('ERRORFILE:'),AT(9,70),USE(?Prompt5)
    STRING(@s40),AT(95,71),USE(ERRORFILEVAR)
END
```

```
CODE
POPERRORS()
ERRORCODEVAR      = ERRORCODE()      !will return 2
ERRORVAR          = ERROR()          !will return "File Not Found"
FILEERRORCODEVAR  = FILEERRORCODE()
FILEERRORVAR      = FILEERRORCODE()
ERRORFILEVAR      = ERRORFILE()      !will return "TESTFILE"
OPEN(WINDOW)
ACCEPT
DISPLAY
END
```

See Also:

PUSHERRORS

PUSHERRORS (write error information)

PUSHERRORS()

PUSHERRORS Writes the last error information to an internal memory stack.

The **PUSHERRORS** procedure writes the current error states of the current thread's `ERRORCODE()`, `ERROR()`, `FILEERRORCODE()`, `FILEERROR()`, `ERRORFILE()`, the C library `errno` variable, and the current Windows error code returned by the `GetLastError()` API function.

`POPERRORS` is used to restore the error states originally saved by **PUSHERRORS**. The main purpose of these functions is to save errors states and then check them later, after performing some additional code that can potentially change them.

PUSHERRORS and `POPERRORS` works using the LIFO (last in, first out) stack convention. The maximum depth of the stack is dependent on available memory. Because `ERRORCODE()`, `ERROR()`, etc. are thread dependent, **PUSHERRORS** and `POPERRORS` statements that are called from different threads are independent of each other, and allocate a different memory stack for each thread.

Example:

```

PROGRAM

MAP
  InitialErrors()
END

TESTFILE   FILE,DRIVER('DOS')
           RECORD
f1         BYTE
           END
           END

CODE
!file processing and initialization here
OPEN(testfile)  !forcing an error
PUSHERRORS()

!additional code here

InitialErrors

```

```
InitialErrors      PROCEDURE

ERRORCODEVAR      LONG
ERRORVAR          CSTRING(255)
FILEERRORCODEVAR  LONG
FILEERRORVAR      CSTRING(255)
ERRORFILEVAR      CSTRING(255)
```

```
Window WINDOW('Error Report'),AT(,,362,128),FONT('MS Sans
Serif',8,,FONT:regular,CHARSET:ANSI),SYSTEM, |
    GRAY
    PROMPT('ERRORCODE:'),AT(9,10),USE(?Prompt1)
    STRING(@N_4),AT(97,10,43,10),USE(ERRORCODEVAR)
    PROMPT('ERROR:'),AT(9,25),USE(?Prompt2)
    STRING(@s40),AT(96,25),USE(ERRORVAR)
    PROMPT('FILEERRORCODE:'),AT(9,40),USE(?Prompt3)
    STRING(@N_4),AT(95,40),USE(FILEERRORCODEVAR)
    PROMPT('FILEERROR:'),AT(9,55),USE(?Prompt4)
    STRING(@s40),AT(95,55),USE(FILEERRORVAR)
    PROMPT('ERRORFILE:'),AT(9,70),USE(?Prompt5)
    STRING(@s40),AT(95,71),USE(ERRORFILEVAR)
END
```

```
CODE
POPERRORS()
ERRORCODEVAR      = ERRORCODE()      !will return 2
ERRORVAR          = ERROR()          !will return "File Not Found"
FILEERRORCODEVAR  = FILEERRORCODE()
FILEERRORVAR      = FILEERRORCODE()
ERRORFILEVAR      = ERRORFILE()      !will return "TESTFILE"
OPEN(WINDOW)
ACCEPT
    DISPLAY
END
```

See Also:

POPERRORS

PUT (re-write record)

```

PUT( | file [, filepointer [, length ] ] | )
      | queue , [[+]key, ..., [-]key] |
      | queue , name |
      | queue , function |
      | view |

```

PUT	Writes a record back to a FILE, QUEUE, or VIEW.
<i>file</i>	The label of a FILE declaration.
<i>filepointer</i>	A numeric constant, variable, or expression for the value returned by the POINTER(<i>file</i>) procedure.
<i>length</i>	An integer constant, variable, or expression containing the number of bytes to write to the <i>file</i> . This must be greater than zero and not greater than the RECORD length. If omitted or out of range, the RECORD length is used.
<i>queue</i>	The label of a QUEUE structure.
+ -	The leading plus or minus sign specifies the <i>key</i> is sorted in ascending or descending sequence.
<i>key</i>	The label of a field declared within the QUEUE structure. If the QUEUE has a PRE attribute, the <i>key</i> must include the prefix.
<i>name</i>	A string constant, variable, or expression containing the NAME attribute of QUEUE fields, separated by commas, and optional leading + or - signs for each attribute. This parameter is case sensitive.
<i>function</i>	The label of the function containing two parameters of a *GROUP or named GROUP passed by address, and having a SIGNED return value. Both parameters must use the same parameter type, and cannot be omitted. The RAW, C and PASCAL attributes are not permitted in the prototype declaration. See Additional Queue Considerations.
<i>view</i>	The label of a VIEW declaration.

The **PUT** statement re-writes a previously accessed record in a FILE, QUEUE, or VIEW.

FILE Usage

The **PUT** statement writes the current values in the RECORD structure data buffer to a previously accessed record in the *file*.

PUT(*file*)

Writes back the last record accessed with NEXT, PREVIOUS, GET, or ADD. If the values in the key variables were changed, the KEYs are updated.

PUT(*file,filepointer*)

Writes the record to the *filepointer* location in the *file* and the KEYs are updated.

PUT(*file,filepointer,length*)

Writes *length* bytes to the *filepointer* location in the *file* and the KEYs are updated.

If a record was not accessed with NEXT, PREVIOUS, GET, REGET, ADD, or was deleted, the "Record Not Available" error is posted. PUT also posts the "Creates Duplicate Key" error. If any error is posted, the record is not written to the file.

QUEUE Usage

PUT writes the contents of the data buffer back to the QUEUE (after a successful GET or ADD) to the position returned by the POINTER procedure. If no previous GET or ADD was executed, the "Entry Not Found" error is posted.

PUT(*queue*)

Writes the data buffer back to the same relative position within the QUEUE of the last successful GET or ADD.

PUT(*queue,key*)

Writes an entry to a sorted memory queue after a successful GET or ADD, maintaining the sort order if any *key* fields have changed value. Multiple *key* parameters may be used (up to 16), separated by commas, with optional leading plus or minus signs to indicate ascending or descending sequence. The entry is inserted immediately after all other entries with matching *key* values.

PUT(*queue,name*)

Writes an entry to a sorted memory queue after a successful GET or ADD, maintaining the sort order if any key fields have changed value. The *name* string must contain the NAME attributes of the fields, separated by commas, with optional leading plus or minus signs to indicate ascending or descending sequence. The entry is inserted immediately after all other entries with matching field values.

PUT(queue,function)

Using PUT by FUNCTION will write from a positional value returned by the function. If the function returns zero (0) the queue record of the first parameter is treated as equal to the second. In this case, no record is written, since the values are equal. If the function returns a negative value, the PUT of the record passed as a first parameter is treated as having less value than record passed as second parameter and is written accordingly. If the function returns a positive value, the PUT of the record passed as a first parameter is treated as having a greater value than record passed as second parameter and is written accordingly.

VIEW Usage

The **PUT** statement writes the current values in the VIEW structure's primary file's data buffer to a previously accessed primary file record in the *view*. If the record was held, it is automatically released. PUT writes to the last record accessed with the REGET, NEXT, or PREVIOUS statements. If the values in the key variables were changed, then the KEYS are updated.

PUT only writes to the primary file in the VIEW because the VIEW structure performs both relational Project and Join operations at the same time. Therefore, it is possible to create a VIEW structure that, if all its component files were updated, would violate the Referential Integrity rules set for the database. The common solution to this problem in SQL-based database products is to write only to the Primary file. Therefore, Clarion has adopted this same industry standard solution.

If a record was not accessed with REGET, NEXT, or PREVIOUS statements, or was deleted, then the "Record Not Available" error is posted. PUT also posts the "Creates Duplicate Key" error. If any error is posted, then the record is not written to disk.

Errors Posted:

- 05 Access Denied
- 08 Insufficient Memory
- 30 Entry Not Found
- 33 Record Not Available
- 40 Creates Duplicate Key
- 75 Invalid Field Type Descriptor
- 89 Record Changed By Another Station

Example:

```
ViewOrder  VIEW(Header)
            JOIN(Dt1:OrderKey,Hea:OrderNumber) !Join Detail file
            PROJECT(Det:Item,Det:Quantity)
            END
            END
```

```
NameQue  QUEUE,PRE(Que)
Name     STRING(20),NAME('FirstField')
Zip      DECIMAL(5,0),NAME('SecondField')
            END
```

```

CODE
OPEN(Header,22h)
OPEN(Detail,22h)
SET(Cus:AcctKey)
OPEN(ViewOrder)
LOOP                                !Read all records in reverse order
  PREVIOUS(ViewOrder)              ! read a record sequentially
  IF ERRORCODE() THEN BREAK. ! break at beginning of file
  DO LastInFirstOut                !Call last in first out routine
  PUT(ViewOrder)                   !Write transaction record back to the file
  IF ERRORCODE() THEN STOP(ERROR()) END
END                                  !End loop

DO BuildQue                         !Call routine to build the queue
Que:Name = 'Jones'                 !Initialize key field
GET(NameQue,Que:Name)             !Get the matching record
  IF ERRORCODE() THEN STOP(ERROR()) END
Que:Zip = 12345                    !Change the zip
PUT(NameQue)                       !Write the changes to the queue
  IF ERRORCODE() THEN STOP(ERROR()) END
Que:Name = 'Jones'                 !Initialize key field
GET(NameQue,Que:Name)             !Get the matching record
  IF ERRORCODE() THEN STOP(ERROR()) END
Que:Name = 'Smith'                !Change key field
PUT(NameQue,Que:Name)             !Write changes to the queue
  IF ERRORCODE() THEN STOP(ERROR()) END
Que:Name = 'Smith'                !Initialize key field
GET(NameQue,'FirstField')         !Get the matching record
  IF ERRORCODE() THEN STOP(ERROR()) END
Que:Name = 'Jones'                !Change key field
PUT(NameQue,'FirstField')         !Write changes to the queue
  IF ERRORCODE() THEN STOP(ERROR()) END

```

See Also:

NEXT

PREVIOUS

GET

ADD

WATCH

HOLD

RELEASE

SORT

PUTINI (set INI file entry)

PUTINI(*section* ,*entry* [,*value*] [,*file*])

PUTINI	Sets the value for an INI file entry.
<i>section</i>	A string constant or variable containing the name of the portion of the INI file which contains the <i>entry</i> .
<i>entry</i>	A string constant or variable containing the name of the specific entry to set.
<i>value</i>	A string constant or variable containing the setting to place in the <i>entry</i> (up to 1023 characters). An empty string (") leaves the <i>entry</i> empty. If omitted, the <i>entry</i> is deleted.
<i>file</i>	A string constant or variable containing the name of the INI file to search (looks for the <i>file</i> in the Windows directory unless a full path is specified). If the file doesn't exist, a new .INI file is created. If omitted, PUTINI places the <i>entry</i> in the WIN.INI file.

The **PUTINI** procedure places the *value* into an *entry* in a Windows-standard .INI file (maximum size of the file is 64K). A Windows-standard .INI file is an ASCII text file with the following format:

```
[some section name]
entry=value
next entry=another value
```

For example, WIN.INI contains entries such as:

```
[windows]
spooler=yes
load=nwpopup.exe
[intl]
sLanguage=enu
sCountry=United States
iCountry=1
```

The PUTINI procedure searches the specified *file* for the *entry* within the *section* you specify. It replaces the current entry value with the *value* you specify. If necessary, the *section* and *entry* are created.

Example:

```
CODE
PUTINI('MyApp','SomeSetting','Initialized')    !Place setting in WIN.INI
PUTINI('MyApp','ASetting','2','MYAPP.INI')    !Place setting in MYAPP.INI
```

See Also:

GETINI

PUTREG (write value to Windows registry)

PUTREG(LONG root, STRING keyname, STRING valuename [, STRING value])

PUTREG	Writes a string value into the system registry.
<i>root</i>	The root section of the registry to which to write the value. Valid values for this are defined in equates.clw and are as follows: REG_CLASSES_ROOT REG_CURRENT_USER REG_LOCAL_MACHINE REG_USERS REG_PERFORMANCE_DATA REG_CURRENT_CONFIG REG_DYN_DATA
<i>Keyname</i>	The key name of the key whose value is to be written. This may contain a path separated by backslash '\' characters.
<i>valuename</i>	The name of the value to be written.
<i>value</i>	The value to be written to the registry in the position given. If omitted, an empty string is written to the registry.

The PUTREG procedure places the value into a valuename that exists in the Windows registry.

Example:

```
PROGRAM

MAP.

INCLUDE('EQUATES')
CurrentPath CSTRING(100)
ColorScheme CSTRING(100)

CODE
CurrentPath = 'C:\Clarion6'

PUTREG(REG_LOCAL_MACHINE, 'SOFTWARE\SoftVelocity\Clarion6', 'root', CurrentPath)
!Sets the root directory of Clarion 6 install
ColorScheme = 'Windows Standard'
PUTREG(REG_CURRENT_USER, 'Control Panel\Current', 'Color
Schemes', ColorScheme)
!writes the current user's color scheme to the registry
```

See Also: GETREG, DELETEREG

QUOTE (replace string special characters)

QUOTE(*string* , *flag*)

QUOTE Expands (or unpacks) *string* data

string A string constant or variable containing the properties to parse.

flag An unsigned integer that controls the method of unpacking.

The way that **QUOTE** processes the *string* is based on the *flag* parameter.

If the *flag* parameter is set to 0 (default), 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. Non-printable characters are replaced with <*n*> sequences, where *n* is a character code. 10 or more repeated characters are replaced with a *a*{*n*} sequence, where *a* is a character and *n* is the number of times that the character is repeated.

If the *flag* parameter is set to 1, **QUOTE** unpacks {} and <> sequences if the next character after the { or < is a numeric digit and doubles all single quotes (').

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:

```
stringvar1  STRING('<250>')
stringvar2  STRING('<display text>')
stringvar3  STRING(30)
```

CODE

```
MESSAGE(QUOTE(stringvar1,0))  !returns accented u
MESSAGE(QUOTE(stringvar1,1))  !returns accented u

MESSAGE(QUOTE(stringvar2,0))  !returns '<<display text>'
MESSAGE(QUOTE(stringvar2,1))  !returns '<display text>'

stringvar3 = 'label{{PROP:text} = value'

MESSAGE(QUOTE(stringvar3,0))  !returns 'label{{PROP:text} = value'
MESSAGE(QUOTE(stringvar3,1))  !returns 'label{PROP:text} = value'
```

See Also:

UNQUOTE

RANDOM (return random number)

RANDOM(*low,high*)

RANDOM Returns random integer.

low A numeric constant, variable, or expression for the lower boundary of the range.

high A numeric constant, variable, or expression for the upper boundary of the range.

The **RANDOM** procedure returns a random integer between the *low* and *high* values, inclusively. The *low* and *high* parameters may be any numeric expression, but only their integer portion is used for the inclusive range.

Return Data Type: LONG

Example:

```
Num          BYTE,DIM(49)
LottoNbr     BYTE,DIM(6)
CODE
CLEAR(Num)
CLEAR(LottoNbr)
LOOP X# = 1 TO 6
  LottoNbr[X#] = RANDOM(1,49)    !Pick numbers for Lotto
  IF NOT Num[LottoNbr[X#]]
    Num[LottoNbr[X#]] = 1
  ELSE
    X# -= 1
  END
END
```

RECORDS (return number of rows in data set)

RECORDS(*entity*)

RECORDS Returns the number of records.

entity The label of a QUEUE, VIEW, FILE, KEY, or INDEX declaration.

The **RECORDS** procedure returns a LONG integer containing the number of entries in the *entity*.

FILE Usage

The RECORDS procedure returns the number of records in a FILE, KEY, or INDEX. Since the OPT attribute of a KEY or INDEX excludes "null" entries, RECORDS may return a smaller number for the KEY or INDEX than the FILE.

QUEUE Usage

The RECORDS procedure returns a LONG integer containing the number of entries in the QUEUE.

VIEW Usage

The RECORDS procedure returns a LONG integer containing the number of rows in the VIEW's return data set, if no KEY fields are used in the VIEW's ORDER attribute.

For non-SQL file systems, if a KEY field is used in the VIEW's ORDER attribute, then RECORDS returns negative one (-1). RECORDS can only return a valid value in the cases where the VIEW engine must build its own index of all the records in the return data set. For those non-SQL VIEWS which do use a KEY field in the ORDER attribute, Clarion's VIEW engine optimizations make use of that KEY (allowing for faster overall processing), so no index is built and the number of records in the return data set is therefore not known.

Return Data Type: LONG

Example:

```
SomeProc PROCEDURE(LocationQueue Location) !receives named QUEUE structure
```

```
Customer FILE,DRIVER('Clarion'),PRE(Cus)
```

```
AcctKey KEY(Cus:AcctNumber)
```

```
NameKey KEY(Cus:Name)
```

```
Record RECORD
```

```
AcctNumber LONG
```

```
Name STRING(20)
```

```

Addr          STRING(20)
CSZ           STRING(60)
              END
              END

```

```

Header FILE,DRIVER('Clarion'),PRE(Hea)
AcctKey KEY(Hea:AcctNumber)
OrderKey KEY(Hea:OrderNumber)
Record RECORD
AcctNumber LONG
OrderNumber LONG
OrderAmount DECIMAL(11,2)
              END
              END

```

```

ViewOrder VIEW(Customer),ORDER('Cus:Name,-Hea:OrderAmount') !ORDER without KEY fields
          PROJECT(Cus:AcctNumber,Cus:Name)
          JOIN(Hea:AcctKey,Cus:AcctNumber)
          PROJECT(Hea:OrderNumber)
          PROJECT(Hea:OrderAmount)
          END
          END

```

```

SaveCount LONG
SaveNameCount LONG
CODE
OPEN(Customer)
OPEN(Header)
SaveCount = RECORDS(Customer)          !Save the record count
SaveNameCount = RECORDS(Cus:NameKey) !Number of records with names filled in
OPEN(ViewOrder)
MESSAGE("Records in VIEW = ' & RECORDS(ViewOrder))

```

```

Entries# = RECORDS(Location)          !Determine number of entries in passed QUEUE
LOOP I# = 1 TO Entries#                !Loop through QUEUE
  GET(Location,I#)                    !getting each entry
  ASSERT(NOT ERRORCODE())
  DO SomeProcess                      !process the entry
END

```

See Also:

QUEUE

ADD

KEY

INDEX

OPT

REGISTER (register event handler)

REGISTER(*event*, *handler*, *object* [, *window*] [, *control*])

REGISTER	Registers an event handling procedure.
<i>event</i>	An integer constant, variable, expression, or EQUATE containing an event number. A value in the range 400h to 0FFFh is a User-defined event.
<i>handler</i>	A LONG variable, or expression containing the return value from ADDRESS for the PROCEDURE to handle the <i>event</i> .
<i>object</i>	A LONG integer constant, variable, or expression containing any 32-bit unique value to identify the specific <i>handler</i> . This is generally the return value of ADDRESS(SELF) when the <i>handler</i> is a CLASS method.
<i>window</i>	The label of the WINDOW or REPORT whose <i>event</i> to handle. If omitted, the current target WINDOW or REPORT is assumed.
<i>control</i>	An integer constant, EQUATE, variable, or expression containing the field number of the specific control whose <i>event</i> to handle. If omitted, the <i>event</i> is handled for every control on the <i>window</i> .

REGISTER registers an event *handler* PROCEDURE called internally by the currently active ACCEPT loop of the specified *window* whenever the specified *event* occurs. This may be a User-defined event, or any other event. User-defined event numbers can be defined as any integer between 400h and 0FFFh.

You may REGISTER multiple *handlers* for the same *event* if you choose--the *handlers* are called by ACCEPT in reverse order of their registration (the last one registered executes first). You may explicitly call UNREGISTER to remove the registration of any specific *handler*. The Clarion runtime library automatically unregisters all registered *event handlers* upon RETURN from the PROCEDURE in which they were registered (when the ACCEPT loop terminates its execution), so explicitly calling UNREGISTER is not required unless your program's logic requires it.

Anytime the *event* occurs, the *handler* procedure is called internally by the currently active ACCEPT loop to process the event. The value returned by the *handler* determines whether or not ACCEPT cycles for any additional *event* processing.

The *handler* PROCEDURE must not take any parameters and must return a BYTE containing one of the following EQUATED values (these EQUATES are defined in the ABERROR.INC file):

Level:Benign

Calls any other *handlers* and the ACCEPT loop, if available.

Level:Notify

Doesn't call other *handlers* or the ACCEPT loop. This is like executing CYCLE when processing the event in an ACCEPT loop.

Level:Fatal

Doesn't call other *handlers* or the ACCEPT loop. This is like executing BREAK when processing the event in an ACCEPT loop.

Example:

```
WindowResizeClass.Init PROCEDURE
CODE
REGISTER(EVENT:Sized,ADDRESS(SELF.TakeResize),ADDRESS(SELF))
!Other code follows
```

```
WindowResizeClass.TakeResize PROCEDURE
ReturnValue BYTE
CODE
ReturnValue = Level:Benign
RETURN(ReturnValue)
```

See Also:

UNREGISTER

ACCEPT

EVENT

REJECTCODE (return reject code number)

REJECTCODE()

The **REJECTCODE** procedure returns the code number for the reason any **EVENT:Rejected** that was posted. If no **EVENT:Rejected** was posted, **REJECTCODE** returns zero. The **EQUATES.CLW** file contains equates for the values returned by **REJECTCODE**:

REJECT:RangeHigh	!Above the top range on a SPIN
REJECT:RangeLow	!Below the bottom range on a SPIN
REJECT:Range	! Other range error
REJECT:Invalid	! Invalid input

Return Data Type: **SIGNED**

Example:

```
CASE EVENT()  
OF EVENT:Rejected  
  EXECUTE REJECTCODE()  
  MESSAGE('Input invalid -- out of range -- too high')  
  MESSAGE('Input invalid -- out of range -- too low')  
  MESSAGE('Input invalid -- out of range')  
  MESSAGE('Input invalid')  
END  
END
```

REGET (re-get record)

REGET(*sequence*, *string*)

REGET	Regets a specific record.
<i>sequence</i>	The label of a VIEW, FILE, KEY, or INDEX declaration.
<i>string</i>	The string returned by the POSITION procedure.

The **REGET** re-reads a previously accessed record.

FILE Usage

REGET reads the record identified by the *string* returned by the POSITION procedure. The value contained in the *string* returned by the POSITION procedure, and its length, are dependent on the file driver.

VIEW Usage

REGET reads the VIEW record identified by the *string* returned by the POSITION(*view*) procedure. The value contained in the *string* returned by the POSITION procedure, and its length, are file driver dependent. If the VIEW contains JOIN structures, REGET retrieves the appropriate set of related records.

REGET re-loads all the VIEW component files' record buffers with complete records. It does not perform the relational "Project" operation. REGET(*view*) is explicitly designed to reset the record buffers to the appropriate records immediately prior to a CLOSE(*view*) statement. However, the processing sequence of the files must be reset with a SET or RESET statement.

Errors Posted:	35	Record Not Found
	37	File Not Open
	43	Record Is Already Held

Example:

```
ViewOrder VIEW(Customer) !Declare VIEW structure
  PROJECT(Cus:AcctNumber,Cus:Name)
  JOIN(Hea:AcctKey,Cus:AcctNumber) !Join Header file
  PROJECT(Hea:OrderNumber)
  JOIN(Dtl:OrderKey,Hea:OrderNumber) !Join Detail file
  PROJECT(Det:Item,Det:Quantity)
  JOIN(Pro:ItemKey,Dtl:Item) !Join Product file
  PROJECT(Pro:Description,Pro:Price)
  END
  END
  END
  END
```

```

RecordQue  QUEUE,PRE(Que)
AcctNumber LIKE(Cus:AcctNumber)
Name       LIKE(Cus:Name)
OrderNumber LIKE(Hea:OrderNumber)
Item       LIKE(Det:Item)
Quantity   LIKE(Det:Quantity)
Description LIKE(Pro:Description)
Price      LIKE(Pro:Price)
SavPosition STRING(260)
          END
CODE
OPEN(Customer,22h)
OPEN((Header,22h)
OPEN(Detail,22h)
OPEN(Product,22h)
SET(Cus:AcctKey)
  OPEN(ViewOrder)
LOOP
  NEXT(ViewOrder)           !Read all records in file
                             !read a record sequentially
  IF ERRORCODE()
    DO DisplayQue
    BREAK
  END
  RecordQue ::= Cus:Record   !Move record into queue
  RecordQue ::= Hea:Record   !Move record into queue
  RecordQue ::= Dtl:Record   !Move record into queue
  RecordQue ::= Pro:Record   !Move record into queue
  SavPosition = POSITION(ViewOrder)!Save record position
  ADD(RecordQue)             !and add it
  ASSERT(NOT ERRORCODE())
END
ACCEPT
CASE ACCEPTED()
OF ?ListBox
  GET(RecordQue,CHOICE())
  REGET(ViewOrder,Que:SavPosition)!Reset the record buffers
  CLOSE(ViewOrder)               !and get the record again
  FREE(RecordQue)
  UpdateProc                       !Call Update Procedure
  BREAK
END
END

```

See Also:

POSITION, SET, RESET, WATCH, GET, NEXT, PREVIOUS

RELEASE (release a held record)

RELEASE(*entity*)

RELEASE Releases the held record(s).

entity The label of a FILE or VIEW declaration.

The **RELEASE** statement releases a record previously held by the HOLD procedure. It will not release a record held by another user in a multi-user environment. If the record is not held, or is held by another user, RELEASE is ignored.

Example:

```
ViewOrder VIEW(Customer) !Declare VIEW structure
    PROJECT(Cus:AcctNumber,Cus:Name)
    JOIN(Hea:AcctKey,Cus:AcctNumber) !Join Header file
    PROJECT(Hea:OrderNumber)
    JOIN(Dtl:OrderKey,Hea:OrderNumber) !Join Detail file
    PROJECT(Det:Item,Det:Quantity)
    JOIN(Pro:ItemKey,Dtl:Item) !Join Product file
    PROJECT(Pro:Description,Pro:Price)
    END
    END
    END
    END
CODE
OPEN(Customer,22h)
OPEN(Header,22h)
OPEN(Detail,22h)
OPEN(Product,22h)
SET(Cus:AcctKey)
OPEN(ViewOrder)
LOOP
    LOOP
        HOLD(ViewOrder,1)
        NEXT(ViewOrder)
        IF ERRORCODE() = 43
            CYCLE
            ELSE;BREAK
        END
    END
    IF ERRORCODE() THEN BREAK END !Check for end of file
    !Process the records
    RELEASE(ViewOrder)
END
```

See Also: HOLD, PUT

REMOVE (erase a file)

REMOVE(*file*)

REMOVE Deletes a FILE.

file The label of the FILE to be removed, or a string constant or variable containing the filename of the file to erase.

The **REMOVE** statement erases a file specification from the operating system directory in the same manner as the DOS "Delete" command. The *file* must be closed, or the "File Already Open" error is posted. If any error is posted, the file is not removed.

Since some file drivers use multiple physical disk files for one logical FILE structure, the default filename and extension assumptions are dependent on the file driver. If any error is posted, the file is not deleted.

Errors Posted: 02 File Not Found
 05 Access Denied
 52 File Already Open

Example:

```
REMOVE(OldFile)            !Delete the old file  
REMOVE('Changes.dat')    !Delete the changes file
```

See Also:

CLOSE

RENAME (change file directory name)

RENAME(*file,new file*)

RENAME	Renames a FILE.
<i>file</i>	The label of a FILE to rename, or a string constant or variable containing a file specification.
<i>new file</i>	A string constant or variable containing a file specification. If the file specification does not contain a drive and path, the current drive and directory are assumed. If only the path is specified, the filename and extension of the original <i>file</i> are used for the <i>new file</i> . Files cannot be renamed to a new drive.

The **RENAME** statement changes the *file* specification to the specification for the *new file* in the directory. The *file* to rename must be closed, or the "File Already Open" error is posted. If the file specification of the *new file* is identical to the original *file*, the RENAME statement is ignored. If any error is posted, the file is not renamed.

Since some file drivers use multiple physical disk files for one logical FILE structure, the default filename and extension assumptions are dependent on the file driver.

Errors Posted:

- 02 File Not Found
- 03 Path Not Found
- 05 Access Denied
- 52 File Already Open

Example:

```

RENAME(Text,'text.bak')           !Make it the backup
RENAME(Master,'\newdir')         !Move it to another directory
RENAME('C:\AUTOEXEC.BAT','C:\AUTOEXEC.SAV') !Make it the backup

```

See Also:

CLOSE

RESET (reset record sequence position)

```
RESET( | view, string | )
      | view, file |
      | sequence, string |
```

RESET	Resets sequential processing to a specific record.
<i>view</i>	The label of a VIEW.
<i>string</i>	The string returned by the POSITION procedure.
<i>file</i>	The label of a component file of the VIEW.
<i>sequence</i>	The label of a FILE, KEY, or INDEX declaration.

RESET resets sequential processing to a specific record.

VIEW Usage

RESET restores the VIEW to a previously read position in the return record set.

RESET(*view, string*)

Resets to the record identified by the *string* that was returned by the POSITION procedure. Once RESET has restored the record pointer, either NEXT or PREVIOUS will read that record.

RESET(*view, file*)

Resets to the record identified by the current contents of the *file's* record buffer. This is used when the order of the VIEW is specified using PROP:Order and is equivalent to a RESET(*view, string*).

The value contained in the *position* string (a value returned by the POSITION procedure) and its length, are file driver dependent. RESET is usually used in conjunction with POSITION to temporarily suspend and resume sequential VIEW processing.

FILE Usage

RESET restores the record pointer to the record identified by the *string* returned by the POSITION procedure. Once RESET has restored the record pointer, either NEXT or PREVIOUS will read that record.

The value contained in the *string* returned by the POSITION procedure, and its length, are dependent on the file driver. RESET is used in conjunction with POSITION to temporarily suspend and resume sequential file processing.

Errors Posted: 33 Record Not Available
37 File Not Open

Example:

```
ViewOrder VIEW(Customer) !Declare VIEW structure
  PROJECT(Cus:AcctNumber,Cus:Name)
  JOIN(Hea:AcctKey,Cus:AcctNumber) !Join Header file
  PROJECT(Hea:OrderNumber)
  JOIN(Dtl:OrderKey,Hea:OrderNumber) !Join Detail file
  PROJECT(Det:Item,Det:Quantity)
  JOIN(Pro:ItemKey,Dtl:Item) !Join Product file
  PROJECT(Pro:Description,Pro:Price)
  END
  END
  END
  END
```

```
RecordQue QUEUE,PRE(Que)
AcctNumber LIKE(Cus:AcctNumber)
Name LIKE(Cus:Name)
OrderNumber LIKE(Hea:OrderNumber)
Item LIKE(Det:Item)
Quantity LIKE(Det:Quantity)
Description LIKE(Pro:Description)
Price LIKE(Pro:Price)
  END
```

```
SavPosition STRING(260)
  CODE
  OPEN(Customer,22h)
  OPEN(Header,22h)
  OPEN(Detail,22h)
  OPEN(Product,22h)
  SET(Cus:AcctKey)
  OPEN(ViewOrder) !Top of file in keyed sequence
  LOOP !Read all records in file
  NEXT(ViewOrder) ! read a record sequentially
  IF ERRORCODE()
  DO DisplayQue
  BREAK
  END
  RecordQue ::= Cus:Record !Move record into queue
  RecordQue ::= Hea:Record !Move record into queue
  RecordQue ::= Dtl:Record !Move record into queue
  RecordQue ::= Pro:Record !Move record into queue
  ADD(RecordQue) ! and add it
  ASSERT(NOT ERRORCODE())
```

```
IF RECORDS(RecordQue) = 20 !20 records in queue?
  DO DisplayQue             !Display the queue
  END
END                           !End loop

DisplayQue ROUTINE
  SavPosition = POSITION(ViewOrder) !Save record position
  DO ProcessQue             !Display the queue
  FREE(RecordQue)         !and free it
  RESET(ViewOrder,SavPosition) !Reset the record pointer
  NEXT(ViewOrder)         !and get the record again
```

See Also:

POSITION, SET, NEXT, PREVIOUS, REGET

RESTORESTATE (restore state of data file)

RESTORESTATE(*file*, *savedstate*, <*restoreflag*>)

RESTORESTATE

Restores a file's *savedstate* state including the record buffer, file pointer, file sequencing order, and the error state of the system.

<i>file</i>	The label of a FILE declaration.
<i>savedstate</i>	A LONG numeric variable which represents the value returned by the GETSTATE procedure.
<i>restoreflag</i>	A BYTE value that sets whether or not the file's record buffer is to be restored. The default value is FALSE (0), which indicates that the file buffer is not restored.

RESTORESTATE restores a file's state including the file pointer, file sequencing order, and the current error state of the system. By default, **RESTORESTATE** does not restore the file's record buffer. Setting the *restoreflag* to TRUE will restore the state of the file's record buffer.

RESTORESTATE does not change the data stored in a file. If data is added to a file after GETSTATE and before RESTORESTATE, the new record(s) will remain in the data file after the RESTORESTATE.

Example:

```
MyFunction PROCEDURE(FILE MyFile)
```

```
CurState LONG
```

```
CODE
```

```
! File will be in the same state on exit of this procedure as it was on entry
CurState = GETSTATE(MyFile)      !save current state of file
SET(MyFile)                       !ready to access file
NEXT(MyFile)                      !read a record
CLEAR(MyFile)                     !clear record buffer
RESTORESTATE(MyFile, CurState, 1) !restore file and buffer to initial state
FREESTATE(MyFile, CurState)       !release resources
```

See Also:

FREESTATE

GETSTATE

RESUME (resume thread execution)

RESUME(*threadno*)

RESUME Resume thread execution.

threadno A numeric constant, variable, or expression that can be evaluated as a SIGNED integer.

The RESUME procedure restores a thread that has been suspended with the SUSPEND statement. If the threadno parameter is a number of a thread that was previously suspended by the call to SUSPEND, its suspending count is decremented. If the suspending count becomes equal to zero(0), execution of the thread continues from the point where it has been suspended. Therefore, the number of calls to RESUME must be equal to the number of calls to SUSPEND for the thread execution to resume.

RESUME can also be used to activate a new thread immediately. Normally, a procedure does not allocate memory for thread variables until the ACCEPT event handler is executed. RESUME can be used to activate a new thread directly upon procedure entry.

Example:

```
RESUME(ThreadNumVariable)    !attempt to resume thread execution
```

```
RESUME(START(MyThreadProc)) !start a new thread immediately
```

See Also: SUSPEND

 Launching a thread - a look behind the scenes

RIGHT (return right justified string)

RIGHT(*string* [,*length*])

RIGHT	Right justifies a string.
<i>string</i>	A string constant, variable, or expression.
<i>length</i>	A numeric constant, variable, or expression for the length of the return string. If omitted, the <i>length</i> is set to the length of the <i>string</i> .

The **RIGHT** procedure returns a right justified string. Trailing spaces are removed, then the string is right justified and returned with leading spaces.

Return Data Type: **STRING**

Example:

```
!RIGHT('ABC ') returns ' ABC'
```

```
Message = RIGHT(Message)    !Right justify the message
```

See Also:

LEFT

CENTER

ROLLBACK (terminate unsuccessful transaction)

ROLLBACK

The **ROLLBACK** statement terminates an active transaction. Execution of a ROLLBACK statement assumes that the transaction was unsuccessful and the database must be restored to the state it was in before the transaction began.

ROLLBACK informs the file driver involved in the transaction that the temporary files containing the information necessary to restore the database to its previous state must be used to restore the database. The file driver then performs the actions necessary to its file system to roll back the transaction.

```
Errors Posted:    65    Unable to ROLLBACK Transaction
                  91    No Logout Active
```

Example:

```
LOGOUT(1,OrderHeader,OrderDetail)    !Begin Transaction
DO ErrHandler                        !always check for errors
ADD(OrderHeader)                     !Add Parent record
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                      !always check for errors
  Det:Record = DetailQue              !Assign to record buffer
  ADD(OrderDetail)                   !and add it to the file
  DO ErrHandler                      !always check for errors
END
COMMIT                               !Terminate successful transaction
ASSERT(~ERRORCODE())

ErrHandler ROUTINE                   !Error routine
IF NOT ERRORCODE() THEN EXIT.        !Bail out if no error
ROLLBACK                             !Rollback the aborted transaction
ASSERT(~ERRORCODE())
BEEP                                  !Alert the user
MESSAGE('Transaction Error - ' & ERROR())
RETURN                                !and get out
```

See Also:

LOGOUT

COMMIT

ROUND (return rounded number)

ROUND(*expression,order*)

ROUND Returns rounded value.

expression A numeric constant, variable, or expression.

order A numeric expression with a value equal to a power of ten, such as 1, 10, 100, 0.1, 0.001, etc. If the value is not an even power of ten, the next lowest power is used; 0.55 will use 0.1 and 155 will use 100.

The **ROUND** procedure returns the value of an *expression* rounded to a power of ten. If the *order* is a LONG or DECIMAL Base Type, then rounding is performed as a BCD operation. Note that if you want to round a real number larger than 1^{30} , you should use `ROUND(num,1.0e0)`, and not `ROUND(num,1)`. The **ROUND** procedure is very efficient ("cheap") as a BCD operation and should be used to compare REALs to DECIMALs at decimal width.

Return Data Type: DECIMAL or REAL

Example:

```
!ROUND(5163,100)    returns 5200
!ROUND(657.50,1)   returns 658
!ROUND(51.63594,.01) returns 51.64
```

```
Commission = ROUND(Price / Rate,.01) !Round the commission to the nearest cent
```

See Also:

BCD Operations and Procedures

ROUNDBOX (draw a box with round corners)

ROUNDBOX(*x* , *y* , *width* , *height* [, *fill*] [, *attributelist*])

ROUNDBOX	Draws a rectangular box with rounded corners on the current window or report.
<i>x</i>	An integer expression that specifies the horizontal position of the starting point.
<i>y</i>	An integer expression that specifies the vertical position of the starting point.
<i>width</i>	An integer expression that specifies the width.
<i>height</i>	An integer expression that specifies the height.
<i>fill</i>	A LONG integer constant, constant EQUATE, or variable containing the red, green, and blue components that create the color in the three low-order bytes (bytes 0, 1, and 2) or an EQUATE for a standard Windows color value.
<i>attributelist</i>	A string constant, variable, or EQUATE containing an optional type of output document and its associated attributes. Only valid when the target is a REPORT. (See <i>EXTEND</i>)

The **ROUNDBOX** procedure places a rectangular box with rounded corners on the current window or report. The position and size of the box are specified by *x*, *y*, *width*, and *height* parameters.

The *x* and *y* parameters specify the starting point, and the *width* and *height* parameters specify the horizontal and vertical size of the box. The box extends to the right and down from its starting point.

The border color is the current pen color set by **SETPENCOLOR**; the default color is the Windows system color for window text. The border width is the current width set by **SETPENWIDTH**; the default width is one pixel. The border style is the current pen style set by **SETPENSTYLE**; the default style is a solid line.

Example:

```
MDIChild WINDOW( 'Child One' ), AT( 0, 0, 320, 200 ), MDI, MAX, HVSCROLL
    !window controls
    END
CODE
OPEN( MDIChild )
ROUNDBOX( 100, 50, 100, 50, 00FF0000h )    !Red round-cornered box
```

See Also: **Current Target**, **SETPENCOLOR**, **SETPENWIDTH**, **SETPENSTYLE**

RUN (execute command)

RUN(*command* [, *waitflag*])

RUN	Executes a <i>command</i> as if it were entered on the DOS command line.
<i>command</i>	A string constant or variable containing the command to execute. This may include a full path and command line parameters.
<i>waitflag</i>	An integer constant, variable, or EQUATE indicating whether RUN should launch the <i>command</i> and wait for its termination, or immediately return after launching. If omitted or zero (0), control immediately returns to the statement following the RUN. If one (1), control returns to the statement following the RUN only after the <i>command</i> has completed its execution.

The **RUN** statement executes a *command* to execute a DOS or Windows program. If the *command* parameter is a STRING variable, you must first use CLIP to remove trailing spaces (not necessary if the *command* is a CSTRING variable). Internally, RUN uses the winexec() Windows API call to execute the *command*.

When the *command* executes, the new program is loaded as the ontop and active program. Execution control in the launching program returns immediately to the statement following RUN and the launching program continues executing as a background application if the *waitflag* is set to zero (0). The user can return to the launching program by either terminating the launched program, or switching back to it through the Windows Task List. Execution control in the launching program returns to the statement following RUN only after the *command* has terminated its execution if the *waitflag* is set to one (1).

If the *command* does not contain a path to the program, the following search sequence is followed:

1. The DOS current directory
2. The Windows directory
3. The Windows system directory
4. Each directory in the DOS PATH
5. Each directory mapped in a network

The successful execution of the *command* may be verified with the RUNCODE procedure, which returns the DOS exit code of the *command*. If unsuccessful, RUN posts the error to the ERROR and ERRORCODE procedures.

Errors Posted: RUN may post any possible error

Example:

```
ProgNameC CSTRING(100)
ProgNameS STRING(100)
```

CODE

```
RUN('notepad.exe readme.txt')      !Run Notepad, automatically loading readme.txt file
RUN(ProgNameC)                     !Run the command in the ProgNameC CSTRING variable
RUN(CLIP(ProgNameS))               !Run the command in the ProgNameS STRING variable
RUN('command.com /c MyBat.bat',1) !Run the command and wait for it to complete
```

See Also: RUNCODE, HALT, ERROR, ERRORCODE

RUNCODE (return program exit code)

RUNCODE()

The **RUNCODE** procedure returns the exit code passed to the operating system from the command executed by the RUN statement. This is the exit code passed by the HALT statement in Clarion programs and is the same as the DOS ERRORLEVEL. RUNCODE returns a LONG integer which may be any value that is returned to DOS as an exit code by the child program.

The child program may only supply a BYTE value as an exit code, therefore negative values are not possible as exit codes. This fact allows RUNCODE to reserve these values to handle situations in which an exit code is not available:

```
0 normal termination
-1 program terminated with Ctrl-C
-2 program terminated with Critical error
-3 TSR exit
-4 program did not run (check ERROR())
```

Return Data Type: LONG

Example:

```
RUN('Nextprog.exe')           !Run next program
IF RUNCODE() = -4
  IF ERROR() = 'Not Enough Memory' !If program didn't run for lack of memory
    MESSAGE('Insufficient memory') !display a message
    RETURN                          !and terminate the procedure
  ELSE
    STOP(ERROR())                  !terminate program
.END.
```

See Also:

RUN

HALT

SELECT (select next control to process)

SELECT([*control*] [,*position*] [,*endposition*])

SELECT	Sets the next control to receive input focus.
<i>control</i>	A field number or field equate label of the next control to process. If omitted, the SELECT statement initiates AcceptAll mode.
<i>position</i>	Specifies a position within the <i>control</i> to place the cursor. For an ENTRY or TEXT, SPIN, or COMBO control this is a character position, or a beginning character position for a marked block. For an OPTION structure, this is the selection number within the OPTION. For a LIST control, this is the QUEUE entry number. This parameter can also be specified using property syntax by PROP:Selected or PROP:SelStart.
<i>endposition</i>	Specifies an ending character position within an ENTRY, TEXT, SPIN, or COMBO <i>control</i> . The character position specified by <i>position</i> and <i>endposition</i> are marked as a block, available for cut and paste operations. This parameter can also be specified using property syntax by PROP:SelEnd.

SELECT overrides the normal TAB key sequence control selection order of an APPLICATION or WINDOW. Its action affects the next ACCEPT statement that executes. The *control* parameter determines which control the ACCEPT loop will process next. If *control* specifies a control which cannot receive focus because a DISABLE or HIDE statement has been issued, focus goes to the next control following it in the window's source code that can receive focus. If *control* specifies a control on a TAB which does not have focus, the TAB is brought to the front before the control receives focus.

SELECT with *position* and *endposition* parameters specifies a marked block in the *control* which is available for cut and paste operations.

SELECT with no parameters initiates AcceptAll mode (also called non-stop mode). This is a field edit mode in which each control in the window is processed in TAB key sequence by generating EVENT:Accepted for each. This allows data entry validation code to execute for all controls, including those that the user has not touched.

AcceptAll mode terminates when any of the following conditions is met:

- A SELECT(?) statement selects the same control for the user to edit. This code usually indicates the value it contains is invalid and the user must re-enter data.
- The Window{PROP:AcceptAll} property is set to zero (0). This property contains one (1) when AcceptAll mode is active. Assigning values to this property can also be used to initiate and terminate AcceptAll mode.

- A control with the REQ attribute is blank or zero. AcceptAll mode terminates with the control highlighted for user entry, without processing any more fields in the TAB key sequence.

When all controls have been processed, EVENT:Completed is posted to the window.

Example:

```
Screen WINDOW,PRE(Scr)
    ENTRY(@N3),USE(Ctl:Code)
    ENTRY(@S30),USE(Ctl:Name)
    LIST,USE(Ctl:Type),From(TypeQue),Drop(5)
    BUTTON('OK'),USE(?OkButton),KEY(EnterKey)
    BUTTON('Cancel'),USE(?CanxButton),KEY(EscKey)
END

CODE
OPEN(Screen)
SELECT(?Ctl:Code)                !Start with Ctl:Code
ACCEPT
CASE SELECTED()
OF ?Ctl:Type
    GET(TypeQue,Ctl:Type)        !Find type in List
    SELECT(?Ctl:Type,POINTER(TypeQue) !Select list to element
END
CASE ACCEPTED()
OF ?Ctl:Code
    IF Ctl:Code > 150            !If data entered is invalid
        BEEP                    !alert the user and
        SELECT(?)              !make them re-enter the data
    END
OF ?Ctl:Name
    SELECT(?Ctl:Name,1,5)       !Mark first five characters as a block
OF ?OkButton
    SELECT                      !Initiate AcceptAll mode
END
IF EVENT() = EVENT:Completed
    BREAK                      !AcceptAll mode terminated
END
END
```

See Also:

ACCEPT

SELECTED (return control that has received focus)

SELECTED()

The **SELECTED** procedure returns the field number of the control receiving input focus when an **EVENT:Selected** event occurs. **SELECTED** returns zero (0) for all other events.

Positive field numbers are assigned by the compiler to all **WINDOW** controls, in the order their declarations occur in the **WINDOW** structure. Negative field numbers are assigned to all **APPLICATION** controls. In executable code statements, field numbers are usually represented by field equate labels--the label of the **USE** variable preceded by a question mark (?FieldName).

Return Data Type: **SIGNED**

Example:

```
CASE SELECTED( )           !Process pre-edit code
OF ?Cus:Company           !Pre-load field value
OF ?Cus:CustType         !Pre-load field value
END
```

See Also:

ACCEPT

SELECT

SEND (send message to file driver)

SEND(*file,message*)

SEND	Sends a message to the file driver.
<i>file</i>	The label of a FILE declaration. The FILE's DRIVER attribute identifies the file driver to receive the <i>message</i> .
<i>message</i>	A string constant or variable containing the information to supply to the file driver.

The **SEND** procedure allows the program to pass any driver-specific information to a file driver during program execution. Valid *messages* are dependent upon the file driver in use. Documentation of all valid SEND *messages* for a given file driver are listed in the file driver's documentation.

Return Data Type: **STRING**

Example:

```
!Arm recovery process for a Clarion data file  
FileCheck = SEND(ClarionFile,'RECOVER=120')
```

SET (initiate sequential file processing)

```

      | file      |
SET( | file, key | )
      | file, filepointer |
      | key      |
      | key, key      |
      | key, keypointer |
      | key, key, filepointer |
      | view      |
      | view , number |

```

SET	Initializes sequential processing of a FILE or VIEW.
<i>file</i>	The label of a FILE declaration. This parameter specifies processing in the physical order in which records occur in the data file.
<i>key</i>	The label of a KEY or INDEX declaration. When used in the first parameter position, <i>key</i> specifies processing in the sort sequence of the KEY or INDEX.
<i>filepointer</i>	A numeric constant, variable, or expression for the value returned by the POINTER(<i>file</i>) procedure.
<i>keypointer</i>	A numeric constant, variable, or expression for the value returned by the POINTER(<i>key</i>) procedure.
<i>view</i>	The label of a VIEW.
<i>number</i>	An integer constant, variable or expression that specifies the start position based on the first <i>number</i> of components of the ORDER attribute. If omitted, all ORDER attribute components are used.

SET initializes sequential processing for a FILE or VIEW. **SET** does not get a record, but only sets up processing order and starting point for the following **NEXT** or **PREVIOUS** statements.

FILE Usage

SET initializes sequential processing of a data file. The first parameter determines the order in which records are processed. The second and third parameters determine the starting point within the file. If the second and third parameters are omitted, processing begins at the beginning (or end) of the file.

```

SET(file)
    Specifies physical record order processing and positions to the beginning
    (SET...NEXT) or end (SET...PREVIOUS) of the file.

```

SET(*file,key*)

Specifies physical record order processing and positions to the first record which contains values matching the values in the component fields of the *key*. NOTE: This form is rarely used and is only useful if the file has been physically sorted in the *key* order. A common mistake is to use this form when SET(*key,key*) is the actual form desired.

SET(*file,filepointer*)

Specifies physical record order processing and positions to the *filepointer* record within the *file*.

SET(*key*)

Specifies keyed sequence processing and positions to the beginning (SET...NEXT) or end (SET...PREVIOUS) of the file in that sequence.

SET(*key,key*)

Specifies keyed sequence processing and positions to the first or last record which contains values matching the values in the component fields of the *key*. Both *key* parameters must be the same.

SET(*key,keypointer*)

Specifies keyed sequence processing and positions to the *keypointer* record within the *key*.

SET(*key,key,filepointer*)

Specifies keyed sequence processing and positions to a record which contains values matching the values in the component fields of the *key* at the exact record number specified by *filepointer*. Both *key* parameters must be the same.

When *key* is the second parameter, processing begins at the first or last record containing values matching the values in all the component fields of the specified KEY or INDEX. If an exact match is found, NEXT will read the first matching record while PREVIOUS will read the last matching record. If no exact match is found, the record with the next greater value is read by NEXT, the record with next lesser value is read by PREVIOUS.

The values for *filepointer* and *keypointer* are dependent on the file driver. They could be a record number, the relative byte position within the file, or some other kind of "seek position" within the file. These parameters are used to begin processing at a specific record within the file.

For all file drivers, an attempt to SET past the end of the file will set the EOF procedure to true, and an attempt to SET before the beginning of the file will set the BOF procedure to true.

VIEW Usage

SET sets sequential processing for the VIEW to the beginning or end of the set of records specified by the FILTER attribute, sorted by the ORDER attribute. The optional *number* parameter limits the SET to assume that the values in the first specified *number* of expressions in the ORDER attribute are fixed. The VIEW must be OPEN before the SET.


```
LOOP                                !Read all records in file
  NEXT(ViewOrder)                  !read a record sequentially
  IF ERRORCODE() THEN BREAK END
  !Process the order
  END
END
Hea:OrderDate = TODAY()-1          !Assign yesterday's date
SET(ViewOrder,1)                  !and process just yesterday's orders
LOOP                                !Read all records in file
  NEXT(ViewOrder)                  !read a record sequentially
  IF ERRORCODE() THEN BREAK.
  !Process the order
  END
END
```

See Also:

NEXT, PREVIOUS, FILTER, ORDER, OPEN, POINTER(queue), POINTER, GET
RESET, POSITION

SET3DLOOK (set 3D window look)

SET3DLOOK(*switch*)

SET3DLOOK Toggles three-dimensional look and feel.

switch An integer constant switching the 3D look off (0) and on (1).

The **SET3DLOOK** procedure sets up the program to display a three-dimensional look and feel. The default program setting is 3D enabled. On a WINDOW, the GRAY attribute causes the controls to display with a three-dimensional appearance. Controls in the TOOLBAR are always displayed with the three-dimensional look, unless disabled by SET3DLOOK. When three-dimensional look is disabled by SET3DLOOK, the GRAY attribute has no effect.

SET3DLOOK(0) turns off the three-dimensional look and feel. SET3DLOOK(1) turns on the three-dimensional look and feel. Values other than zero or one are reserved for future use.

Example:

```

MainWin APPLICATION('My Application'),SYSTEM,MAX,ICON('MyIcon.ICO'),STATUS,|
    HVSCROLL,RESIZE
    MENUBAR
        MENU('&File'),USE(?FileMenu)
        ITEM('&Open...'),USE(?OpenFile)
        ITEM('&Close'),USE(?CloseFile),DISABLE
        ITEM('Turn off 3D Look'),USE(?Toggle3D),CHECK
        ITEM('E&xit'),USE(?MainExit)
    END
END
END
CODE
OPEN(MainWin)
ACCEPT
CASE ACCEPTED()
OF ?Toggle3D
    IF MainWin$?Toggle3D{PROP:text} = 'Turn off 3D Look'    !If on
        SET3DLOOK(0)                                        !Turn off
        MainWin$?Toggle3D{PROP:text} = 'Turn on 3D Look'    !and change text
    ELSE                                                    !Else
        SET3DLOOK(1)                                        !Turn on
        MainWin$?Toggle3D{PROP:text} = 'Turn off 3D Look'    ! and change text
    END
OF ?OpenFile
    START(OpenFileProc)
OF ?MainExit
    BREAK
END
END; CLOSE(MainWin)

```

SETCLIPBOARD (set windows clipboard contents)

SETCLIPBOARD(*string*)

SETCLIPBOARD Puts information in the Windows clipboard.

string A string constant or variable containing the information to place in the Windows clipboard. This should not contain any embedded null characters (ASCII 0). This is placed in the CF_TEXT format, only.

The **SETCLIPBOARD** procedure places the contents of the *string* into the Windows clipboard, overwriting any previous contents.

Example:

```

Que1 QUEUE
  STRING(30)
  END
Que2 QUEUE
  STRING(30)
  END
WinOne WINDOW,AT(0,0,160,400)
  LIST,AT(120,0,20,20),USE(?List1),FROM(Que1),DRAGID('List1')
      !Allows drags, but not drops
  LIST,AT(120,120,20,20),USE(?List2),FROM(Que2),DROPID('List1')
      !Allows drops from List1, but no drags
  END
CODE
OPEN(WinOne)
ACCEPT
CASE EVENT()
OF EVENT:Drag              !When a drag event is attempted
  IF DRAGID()              !check for success
    SETCLIPBOARD(Que1)    !and setup info to pass
  END
OF EVENT:Drop              !When drop event is successful
  Que2 = CLIPBOARD()      !get dropped info
  ADD(Que2)                !and add it to the queue
END
END

```

See Also:

CLIPBOARD

SETCLOCK (set system time)

SETCLOCK(*time*)

SETCLOCK Sets the DOS system clock.

time A numeric constant, variable, or expression for a standard time (expressed as hundredths of a second since midnight, plus one).

The **SETCLOCK** statement sets the operating system time of day.

Example:

```
SETCLOCK(1)      !Set clock to midnight
SETCLOCK(6001)   !Set clock to one minute past midnight
```

See Also:

Standard Time

CLOCK

SETCOMMAND (set command line parameters)

SETCOMMAND(*commandline*)

SETCOMMAND Internally sets command line parameters.

commandline A string constant, variable, or expression containing the new command line parameters.

SETCOMMAND allows the program to internally specify command line parameters that may be read by the COMMAND procedure. SETCOMMAND overwrites any previous command line flag of the same value. To turn off a leading slash flag, append an equal sign (=) to it in the *commandline*.

SETCOMMAND may not be used to set system level switches which must be specified when the program is loaded. The temporary files directory switch (CLATMP=) may be set with SETCOMMAND.

Example:

```
SETCOMMAND(' /N' )      !Add /N parameter
SETCOMMAND(' /N=' )    !Turn off /N parameter
```

See Also:

COMMAND

SETCURSOR (set temporary mouse cursor)

SETCURSOR(*cursor*)

SETCURSOR Specifies a temporary mouse cursor to display.

cursor An EQUATE naming a Windows-standard mouse cursor, or a string constant naming a cursor resource linked into the project--the name of a .CUR file with a leading tilde ('~Mycur.CUR'). If omitted, turns off the temporary cursor.

The **SETCURSOR** statement specifies a temporary mouse *cursor* to display until a SETCURSOR statement without a *cursor* parameter turns it off. This cursor overrides all CURSOR attributes. When SETCURSOR without a *cursor* parameter is encountered, all CURSOR attributes once again take effect. SETCURSOR is generally used to display the hourglass while your program is doing some "behind the scenes" work that the user should not break into.

EQUATE statements for the Windows-standard mouse cursors are contained in the EQUATES.CLW file. The following list is a representative sample of these (see EQUATES.CLW for the complete list):

CURSOR:None	No mouse cursor
CURSOR:Arrow	Normal windows arrow cursor
CURSOR:IBeam	Capital "I" like a steel I-beam
CURSOR:Wait	Hourglass
CURSOR:Cross	Large plus sign
CURSOR:UpArrow	Vertical arrow
CURSOR:Size	Four-headed arrow
CURSOR:Icon	Box within a box
CURSOR:SizeNWSE	Double-headed arrow slanting left
CURSOR:SizeNESW	Double-headed arrow slanting right
CURSOR:SizeWE	Double-headed horizontal arrow
CURSOR:SizeNS	Double-headed vertical arrow

Example:

```

MainWin APPLICATION('My Application'),SYSTEM,MAX,ICON('MyIcon.ICO'),STATUS,HVSCROLL
    MENUBAR
        ITEM('Batch Update'),USE(?Batch)
    END
END

CODE
OPEN(MainWin)
ACCEPT
CASE ACCEPTED()
OF ?Batch
    SETCURSOR(CURSOR:Wait)    !Turn on hourglass mouse cursor
    BatchUpdate              !and call the batch update procedure
    SETCURSOR                !then turn off hourglass
END; END !CASE and ACCEPT

```

SETDROPID (set DROPID return string)

SETDROPID(*string*)

SETDROPID Sets the DROPID procedure's return value.

string A string constant or variable containing the value the DROPID procedure will return.

The **SETDROPID** procedure sets the DROPID procedure's return value. This allows the DROPID procedure to pass the data in a drag-and-drop operation. When drag-and-drop operations are performed between separate Clarion applications, this is the mechanism to use to pass the data.

Example:

```

Que1 QUEUE
    STRING(30)
    END
Que2 QUEUE
    STRING(30)
    END

WinOne WINDOW,AT(0,0,160,400)
    LIST,AT(120,0,20,20),USE(?List1),FROM(Que1),DRAGID('List1')
        !Allows drags, but not drops
    LIST,AT(120,120,20,20),USE(?List2),FROM(Que2),DROPID('List1')
        !Allows drops from List1 or the Windows File Manager,
        !but no drags

    END

CODE
OPEN(WinOne)
ACCEPT
CASE EVENT()
OF EVENT:Drag      !When a drag event is attempted
    IF DRAGID()    !check for success
        SETDROPID(Que1) !and setup info to pass
    END
OF EVENT:Drop      !When drop event is successful
    Que2 = DROPID() !get dropped info, from List1 or File Manager
    ADD(Que2)       !and add it to the queue
END
END
END

```

See Also:

DRAGID

DROPID

SETFONT (specify font)

SETFONT(*control* , [*typeface*] , [*size*] , [*color*] , [*style*] , [*charset*])

SETFONT	Dynamically sets the display font for a control.
<i>control</i>	A field number or field equate label for the control to affect. If <i>control</i> is zero (0), it specifies the WINDOW.
<i>typeface</i>	A string constant or variable containing the name of the font. If omitted, the system font is used.
<i>size</i>	An integer constant or variable containing the size (in points) of the font. If omitted, the system default font size is used.
<i>color</i>	A LONG integer constant or variable containing the red, green, and blue values for the color of the font in the low-order three bytes, or an EQUATE for a standard Windows color value. If omitted, black is used.
<i>style</i>	An integer constant, constant expression, EQUATE, or variable specifying the strike weight and style of the font. If omitted, the weight is normal.
<i>charset</i>	A LONG integer variable specifying the character set value.

SETFONT dynamically specifies the display font for the *control*, overriding any FONT attribute. If the *control* parameter is zero (0), SETFONT specifies the default font for the window. However, this does not affect existing controls--only controls CREATED after SETFONT executes are affected.

SETFONT allows you to specify all parameters of a font change at once, instead of one at a time as runtime property assignment allows. This has the advantage of implementing all changes at once, whereas runtime property assignment would change each individually, displaying each separate change as it occurs.

The *typeface* may name any font registered in the Windows system. The EQUATES.CLW file contains EQUATE values for standard *style* values. A *style* on the range zero (0) to one thousand (1000) specifies the strike weight of the font. You may also add values that indicate italic, underline, or strikeout text. The following EQUATES are in EQUATES.CLW:

FONT:thin	EQUATE (100)
FONT:regular	EQUATE (400)
FONT:bold	EQUATE (700)
FONT:italic	EQUATE (01000H)
FONT:underline	EQUATE (02000H)
FONT:strikeout	EQUATE (04000H)

Example:

```
SETFONT(1, 'Arial', 14, , FONT:thin+FONT:Italic)    !14 pt. Arial black thin italic
```

See Also: GETFONT

SETKEYCHAR (specify ASCII code)

SETKEYCHAR(*keychar*)

SETKEYCHAR Sets the ASCII character returned by the KEYCHAR procedure.

keychar An integer constant, variable, or expression containing the ASCII value of the character to set.

SETKEYCHAR sets the internal ASCII character returned by the KEYCHAR procedure. The character is not put into the keyboard buffer.

Example:

```
SETKEYCHAR(VAL('A'))            !Set up the keychar procedure to return 'A'
```

See Also:

KEYCHAR

SETKEYCODE (specify keycode)

SETKEYCODE(*keycode*)

SETKEYCODE Sets the keycode returned by the KEYCODE procedure.

keycode An integer constant or keycode EQUATE label.

SETKEYCODE sets the internal keycode returned by the KEYCODE procedure. The keycode is not put into the keyboard buffer.

Example:

```
SETKEYCODE(0800h)            !Set up the keycode procedure to return 0800h
```

See Also:

KEYCODE

Keycode Equate Labels

SETNONULL (set file field non-null)

SETNONULL(*field*)

SETNONULL Assigns non-null value (blank or zero) to a *field*.

field The label (including prefix) of a field in a FILE structure. This may be a GROUP or RECORD structure.

The **SETNONULL** statement assigns a non-null value (blank or zero) to a *field* in a FILE structure. If the *field* is a GROUP or RECORD structure, all component fields are set non-null. Support for null "values" in a FILE is entirely dependent upon the file driver.

Return Data Type: LONG

Example:

```
Customer  FILE,DRIVER('Clarion'),PRE(Cus)           !Declare customer file layout
AcctKey   KEY(Cus:AcctNumber)
Record    RECORD
AcctNumber LONG
OrderNumber LONG
Name      STRING(20)
Addr      STRING(20)
          END
          END
Header    FILE,DRIVER('Clarion'),PRE(Hea)         !Declare header file layout
AcctKey   KEY(Hea:AcctNumber)
OrderKey  KEY(Hea:OrderNumber)
Record    RECORD
AcctNumber LONG
OrderNumber LONG
ShipToName STRING(20)
ShipToAddr STRING(20)
          END
          END
```


SETNULL (set file field null)

SETNULL(*field*)

SETNULL Assigns null "value" to a *field*.

field The label (including prefix) of a field in a FILE structure. This may be a GROUP or RECORD structure.

The **SETNULL** statement assigns a null "value" to a *field* in a FILE structure. If the *field* is a GROUP or RECORD structure, all component fields are set to null. Support for null "values" in a FILE is entirely dependent upon the file driver.

Return Data Type: LONG

Example:

```
Customer FILE,DRIVER('Clarion'),PRE(Cus) !Declare customer file layout
AcctKey   KEY(Cus:AcctNumber)
Record    RECORD
AcctNumber LONG
OrderNumber LONG
Name      STRING(20)
Addr      STRING(20)
CSZ       STRING(35)
          END
          END

Header FILE,DRIVER('Clarion'),PRE(Hea) !Declare header file layout
AcctKey   KEY(Hea:AcctNumber)
OrderKey  KEY(Hea:OrderNumber)
Record    RECORD
AcctNumber LONG
OrderNumber LONG
ShipToName STRING(20)
ShipToAddr STRING(20)
ShipToCSZ STRING(35)
          END
          END
```

```
CODE
OPEN(Header)
OPEN(Customer)
SET(Hea:AcctKey)
LOOP
  NEXT(Header)
  IF ERRORCODE( )
    BREAK
  END
  Cus:AcctNumber = Hea:AcctNumber
  GET(Customer,Cus:AcctKey)                                !Get Customer record

IF ERRORCODE( )
  CLEAR(Cus:Record)
  END
  IF NOT NULL(Hea:ShipToName) AND Hea:ShipToName = Cus:Name !Check ship-to address
  SETNULL(Hea:ShipToName)                                   !and assign null "values"
  SETNULL(Hea:ShipToAddr)                                  ! to ship-to address
  SETNULL(Hea:ShipToCSZ)
  END
  PUT(Header)                                              !Put Header record back
END
```

See Also:

NULL

SETNONNULL

SETNULLS(set the null state of columns)

SETNULLS(*table*, *null-value*)

SETNULLS Set NULL state of columns in a table.

table The label of a FILE.

null-value Null "value" to assign to the table.

SETNULLS sets the NULL state of every column in the table based on the contents of the passed string. The passed string must be obtained from a prior call to GETNULLS. This function is valid on all SQL table types.

Errors Posted: 75 Invalid Column Type Descriptor – Indicating that the passed string was too small.
 80 Function Not Supported

Example:

```
StorageString STRING(255)
Rec           STRING(2048)
CODE
StorageString = GETNULLS(table)  !Get NULL state of table
Rec = table.record              !Save record buffer
! Do Table Processing
Table.Record = Rec              !Restore record buffer
SETNULLS(table, StorageString)  !Set NULL state of table columns
```

See Also:

GETNULLS

NULL

SETNONULL

SETPATH (change current drive and directory)

SETPATH(*path*)

SETPATH Changes the current drive and directory.

path A string constant or the label of a STRING, CSTRING, or PSTRING variable containing a new drive and/or directory specification.

SETPATH changes the current drive and directory. If the *drive and path* entry is invalid, the "Path Not Found" error is posted, and the current directory is not changed.

If the drive letter and colon are omitted from the *path*, the current drive is assumed. If only a drive letter and colon are in the *path*, SETPATH changes to the current directory of that drive.

Errors Posted: 03 Path Not Found

Example:

```
SETPATH('C:\LEDGER')      !Change to the ledger directory
SETPATH(UserPath)         !Change to the user's directory
```

See Also:

PATH

SHORTPATH

LONGPATH

DIRECTORY

SETPENCOLOR (set line draw color)

SETPENCOLOR(*color*)

SETPENCOLOR	Sets the current pen color.
<i>color</i>	A LONG integer constant, constant EQUATE, or variable containing the red, green, and blue components that create the color in the three low-order bytes (bytes 0, 1, and 2) or an EQUATE for a standard Windows color value. If omitted, the Windows system color for window text is set.

The **SETPENCOLOR** procedure sets the current pen color for use by all graphics procedures. The default color is the Windows system color for window text.

Every window has its own current pen color. Therefore, to consistently use the same pen (which does not use the default color setting) across multiple windows, the SETPENCOLOR statement should be issued for each window.

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
    !window controls
    END
CODE
OPEN(MDIChild)
SETPENCOLOR(000000FFh)           !Set blue pen color
ROUNDBOX(100,50,100,50,00FF0000h) !Red round-cornered box with blue border
```

See Also:

PENCOLOR

SETPENSTYLE (set line draw style)

SETPENSTYLE([*style*])

SETPENSTYLE Sets the current pen style.

style An integer constant, constant EQUATE, or variable that specifies the pen's style. If omitted, a solid line is set.

The **SETPENSTYLE** procedure sets the current line draw style for use by all graphics procedures. The default is a solid line.

Every window has its own current pen style. Therefore, to consistently use the same pen (which does not use the default style setting) across multiple windows, the SETPENSTYLE statement should be issued for each window.

EQUATE statements for the pen styles are contained in the EQUATES.CLW file. The following list is a representative sample of these (see EQUATES.CLW for the complete list):

PEN:solid	Solid line
PEN:dash	Dashed line
PEN:dot	Dotted line
PEN:dashdot	Mixed dashes and dots

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
  !window controls
  END
CODE
OPEN(MDIChild)
SETPENCOLOR(000000FFh)           !Set blue pen color
SETPENSTYLE(PEN:dash)           !Set dashes for line style
ROUNDBOX(100,50,100,50,00FF0000h) !Red round-cornered box with blue dashed border
```

See Also:

PENSTYLE

SETPENWIDTH (set line draw thickness)

SETPENWIDTH([*width*])

SETPENWIDTH	Sets the current pen width.
<i>width</i>	An integer expression that specifies the pen's thickness, measured in dialog units (unless overridden by the THOUS, MM, or POINTS attribute on a REPORT). If omitted, the default (one pixel) is set.

The **SETPENWIDTH** procedure sets the current line draw thickness for use by all graphics procedures. The default is one pixel, which may be set with a *width* of zero (0).

Every window has its own current pen width. Therefore, to consistently use the same pen (which does not use the default width setting) across multiple windows, the SETPENWIDTH statement should be issued for each window.

Example:

```
MDIChild WINDOW( 'Child One' ), AT(0,0,320,200), MDI, MAX, HVSCROLL
    !window controls
    END

CODE
OPEN(MDIChild)
SETPENCOLOR(000000FFh)      !Set blue pen color
SETPENSTYLE(PEN:dash)      !Set dashes for line style
SETPENWIDTH(2)             !Set two dialog unit thickness
BOX(100,50,100,50,00FF0000h) !Red box with thick blue dashed border
```

See Also:

PENWIDTH

SETPOSITION (specify new control position)

SETPOSITION(*control* [, *x*] [, *y*] [, *width*] [, *height*])

SETPOSITION Dynamically specifies the position and size of an APPLICATION, WINDOW, or control.

<i>control</i>	A field number or field equate label for the control to affect. If <i>control</i> is zero (0), it specifies the window.
<i>x</i>	An integer constant, expression, or variable that specifies the horizontal position of the top left corner. If omitted, the <i>x</i> position is not changed.
<i>y</i>	An integer constant, expression, or variable that specifies the vertical position of the top left corner. If omitted, the <i>y</i> position is not changed.
<i>width</i>	An integer constant, expression, or variable that specifies the width. If omitted, the <i>width</i> is not changed.
<i>height</i>	An integer constant, expression, or variable that specifies the height. If omitted, the <i>height</i> is not changed.

SETPOSITION dynamically specifies the position and size of an APPLICATION, WINDOW, or control. If any parameter is omitted, the value is not changed.

The values contained in the *x*, *y*, *width*, and *height* parameters are measured in dialog units. Dialog units are defined as one-quarter the average character width by one-eighth the average character height. The size of a dialog unit is dependent upon the size of the default font for the window. This measurement is based on the font specified in the FONT attribute of the window, or the system default font specified by Windows.

Using SETPOSITION produces a "smoother" control appearance change than using property expressions to change the AT attribute's parameter values. This is because SETPOSITION changes all four parameters at once. Property expressions must change one parameter at a time. Since each individual parameter change would be immediately visible on screen, this would cause the control to appear to "jump."

Example:

```
CREATE(?Code4Entry,CREATE:entry,?Ctl:Code) !Create a control
?Code4Entry{PROP:use} = 'Code4Entry'      !Set USE variable
?Code4Entry{PROP:text} = '@s10'          !Set entry picture
GETPOSITION(?Ctl:Code,X,Y,Width,Height)  !Get Ctl:Code position
SETPOSITION(?Code4Entry,X+Width+40,Y)    !Set x 40 past Ctl:Code
UNHIDE(?Code4Entry)                      !Display the new control
```

See Also:

GETPOSITION

SETTARGET (set current window or report)

SETTARGET([| *target* |])

| , *thread* |

| *target* , *band* |

SETTARGET	Sets the current window (or report) for drawing graphics and other window-interaction statements.
<i>target</i>	The label of an APPLICATION, WINDOW, or REPORT structure, or a reference to any of those structures. The execution <i>thread</i> is always deduced from the <i>target</i> and any specified <i>thread</i> parameter is ignored. If omitted, the last window opened and not yet closed in the specified <i>thread</i> is used.
<i>thread</i>	The number of the execution thread whose topmost procedure contains the window to set as the <i>target</i> . If omitted, the execution <i>thread</i> is deduced from the <i>target</i> .
<i>band</i>	The control number or field equate label of the REPORT band (or IMAGE control in a window <i>target</i>) to draw graphics primitives to (ARC, CHORD, etc.).

SETTARGET sets the current *target* for runtime property assignment, and the CREATE, SETPOSITION, GETPOSITION, SETFONT, GETFONT, DISABLE, HIDE, CONTENTS, DISPLAY, ERASE, and UPDATE statements. Using these statements with SETTARGET allows you to manipulate the window display in the topmost window of any execution thread. SETTARGET also specifies the *target* structure for drawing with the graphics primitives procedures (ARC, CHORD, etc.).

SETTARGET sets the "built-in" variable TARGET (also set when a window opens), which may be used in any statement which requires the label of the current window or report. SETTARGET does not change procedures, and does not change which ACCEPT loop receives the events generated by Windows. SETTARGET without any parameters resets to the procedure and execution thread with the currently active ACCEPT loop.

A REPORT structure is never the default TARGET. Therefore, SETTARGET must be used before using the graphics primitives procedures to draw graphics on a REPORT. To draw graphics to a specific band in the REPORT (or onto an IMAGE in a window), you must specify the *band* as the second parameter.

SETTARGET

Resets TARGET to the topmost window in the execution thread with the currently active ACCEPT loop.

SETTARGET(*target*)

Sets TARGET to the specified window or report. The execution *thread* is deduced from the *target*.

SETTARGET(*target,thread*)

Sets TARGET to the specified window or report. The execution *thread* is deduced from the *target* parameter and any specified *thread* parameter is ignored.

SETTARGET(, *thread*)

Sets TARGET to the topmost window in the specified execution *thread*.

SETTARGET(*target,band*)

Sets TARGET to the specified *target* window or report, and draws graphics primitives to the specified *band* (report band or IMAGE control).

Example:

```
Report  REPORT
        HEADER,USE(?PageHeader)
        END
Detail  DETAIL
        END
        END
CODE
OPEN(Report)
SETTARGET(Report,?PageHeader)  !Make the report the current target
TARGET{PROP:Landscape} = 1     !and turn on landscape mode
ARC(100,50,100,50,0,900)       !Draw 90 degree arc from 3 to 12 o'clock, as
                                !the top-right quadrant of ellipse
                                !to the page HEADER band

SETTARGET                       !Reset to top window
```

See Also:

START

THREAD

SETTODAY (set system date)

SETTODAY(*date*)

SETTODAY Sets the DOS system date.

date A numeric constant, variable, or expression for a standard date.

The **SETTODAY** statement sets the operating system date.

Example:

```
SETTODAY(TODAY() + 1)    !Set the date ahead one day
```

See Also:

Standard Date

DAY

MONTH

YEAR

TODAY

DATE

SHORTPATH (return short filename)

SHORTPATH([*longfilename*])

SHORTPATH Returns the fully-qualified short filename for a given long filename.

longfilename A string constant, variable, or expression that specifies the long filename to convert. This may include the complete path. If omitted, **SHORTPATH** returns the current drive and directory in short name form.

The **SHORTPATH** procedure returns the DOS standard short filename for a given *longfilename*. The file named as the parameter must exist on disk. **SHORTPATH** can now be used in expressions evaluated by the **EVALUATE** statements in runtime applications and in the templates.

Return Data Type: **STRING**

Example:

```
MyFile STRING(64)
```

```
CODE
```

```
MyFile = SHORTPATH('c:\program files\my text file.txt') !c:\progra~1\mytext~1.txt
```

See Also:

SETPATH

LONGPATH

PATH

DIRECTORY

SHOW (write to screen)

SHOW(*x*, *y*, *string*)

SHOW	Writes a <i>string</i> to the current window or report.
<i>x</i>	An integer expression that specifies the horizontal position of the starting point, in dialog units.
<i>y</i>	An integer expression that specifies the vertical position of the starting point, in dialog units.
<i>string</i>	A string constant, variable, or expression containing the formatted text to place on the current window or report.

SHOW writes the *string* text to the current window or report. The font used is the current font for the window or report.

Example:

```
MDIChild WINDOW( 'Child One' ), AT( 0, 0, 320, 200 ), MDI, MAX, HVSCROLL
    !window controls
    END
CODE
OPEN( MDIChild )
DISPLAY
SHOW( 100, 100, FORMAT( TODAY(), @D3 ) )      !Display the date
SHOW( 20, 20, 'Press Any Key to Continue' )  !Display a message
```

See Also:

Current Target

SHUTDOWN (arm termination procedure)

SHUTDOWN(*[procedure]*)

SHUTDOWN Arms a procedure which is called when the program terminates.

procedure The label of a PROCEDURE. If omitted, the **SHUTDOWN** process is disarmed.

The **SHUTDOWN** statement arms a *procedure* which is called when the program terminates. The shutdown *procedure* is called by normal program termination or by an abnormal-end/run-time halt. It may not be able to execute for an abnormal-end/run-time halt, depending upon the state of the system resources at the time of the crash. It is not called if the computer is rebooted or the program is terminated due to power failure. The same effect as SHUTDOWN can be more safely achieved by simply calling a procedure to execute on EVENT:CloseDown for the application frame.

Example:

```
SHUTDOWN(CloseSys)           !Arm CloseSys as the shutdown procedure
```

See Also:

HALT

RETURN

SIN (return sine)

SIN(*radians*)

SIN	Returns sine.
<i>radians</i>	A numeric constant, variable or expression for the angle expressed in radians. π is a constant which represents the ratio of the circumference and radius of a circle. There are 2π radians (or 360 degrees) in a circle.

The **SIN** procedure returns the trigonometric sine of an angle measured in *radians*. The sine is the ratio of the length of the angle's opposite side divided by the length of the hypotenuse.

Return Data Type: REAL

Example:

```

PI          EQUATE(3.1415926535898)      !The value of PI
Rad2Deg     EQUATE(57.295779513082)     !Number of degrees in a radian
Deg2Rad     EQUATE(0.0174532925199)     !Number of radians in a degree
CODE
  Angle = 45 * Deg2Rad                  !Translate 45 degrees to Radians
  SineAngle = SIN(Angle)                !Get the sine of 45 degree angle

```

See Also:

TAN

ATAN

ASIN

COS

ACOS

SKIP (bypass records in sequence)

SKIP(*entity*, *count*)

SKIP	Bypasses records during sequential processing.
<i>entity</i>	The label of a FILE or VIEW declaration.
<i>count</i>	A numeric constant or variable. The <i>count</i> specifies the number of records to bypass. If the value is positive, records are skipped in forward (NEXT) sequence; if <i>count</i> is negative, records are skipped in reverse (PREVIOUS) sequence.

The **SKIP** statement is used to bypass records during sequential processing. It bypasses records, in the sequence specified by the SET statement, by moving the file pointer *count* records. SKIP is more efficient than NEXT or PREVIOUS for skipping past records because it does not move records into the RECORD structure data buffer. If SKIP reads past the end or beginning of file, the EOF() and BOF() procedures return true. If no SET has been issued, SKIP posts error 33 (Record Not Available).

Errors Posted: 33 Record Not Available
 37 File Not Open

Example:

```
ViewOrder  VIEW(Customer)                !Declare VIEW structure
            PROJECT(Cus:AcctNumber,Cus:Name)
            JOIN(Hea:AcctKey,Cus:AcctNumber) !Join Header file
            PROJECT(Hea:OrderNumber)
            JOIN(Dtl:OrderKey,Hea:OrderNumber) !Join Detail file
            PROJECT(Det:Item,Det:Quantity)
            JOIN(Pro:ItemKey,Dtl:Item)       !Join Product file
            PROJECT(Pro:Description,Pro:Price)
            END
            END
            END
            END
SavOrderNo LONG
CODE
OPEN(Customer,22h)
OPEN((Header,22h)
OPEN(Detail,22h)
OPEN(Product,22h)
SET(Cus:AcctKey)
OPEN(ViewOrder)                !Top of file in keyed sequence
LOOP                            !Process all records
  NEXT(ViewOrder)              !Get a record
  IF ERRORCODE() THEN BREAK END
  IF Hea:OrderNumber <> SavOrderNo !Check for first item in order
```

```
IF Hea:OrderStatus = 'Cancel'           !Is it a canceled order?
  SKIP(Items,Vew:ItemCount-1)           !SKIP rest of the items
  CYCLE                                  !and process next order
  END                                     !end ifs
END                                       !end ifs
DO ItemProcess                           !process the item
  SavInvNo = Hea:OrderNumber            !save the invoice number
END                                       !End loop
```

See Also: SET, RESET, NEXT, PREVIOUS

SORT (sort queue entries)

```
SORT(queue, | [+]key,...,[-]key | )
      | name |
      | function |
```

SORT	Reorders entries in a QUEUE.
<i>queue</i>	The label of a QUEUE structure, or the label of a passed QUEUE parameter.
+ -	The leading plus or minus sign specifies the <i>key</i> will be sorted in ascending or descending sequence.
<i>key</i>	The label of a field declared within the QUEUE structure. If the QUEUE has a PRE attribute, the <i>key</i> must include the prefix. This may not be a reference variable.
<i>name</i>	A string constant, variable, or expression containing the NAME attribute of QUEUE fields, separated by commas, and optional leading + or - signs for each attribute. This parameter is case sensitive and may not contain any reference variables.
<i>function</i>	The label of the function containing two parameters of a *GROUP or named GROUP passed by address, and having a SIGNED return value. Both parameters must use the same parameter type, and cannot be omitted. The RAW, C and PASCAL attributes are not permitted in the prototype declaration. See Additional Queue Considerations.

SORT reorders the entries in a QUEUE. QUEUE entries with identical key values maintain their relative position.

SORT(*queue*,*key*)

Reorders the QUEUE in the sequence specified by the *key*. Multiple *key* parameters may be used (up to 16), separated by commas, with optional leading plus or minus signs to indicate ascending or descending sequence.

SORT(*queue*,*name*)

Reorders the QUEUE in the sequence specified by the *name* string. The *name* string must contain the NAME attributes of the fields, separated by commas, with leading plus or minus signs to indicate ascending or descending sequence.

Errors Posted: 08 Insufficient Memory
 75 Invalid Field Type Descriptor

Example:

```
Location  QUEUE,PRE(Loc)
Name      STRING(20),NAME('FirstField')
City      STRING(10),NAME('SecondField')
State     STRING(2)
Zip       DECIMAL(5,0)
          END

CODE
SORT(Location,Loc:State,Loc:City,Loc:Zip)      !Sort by zip in city in state
SORT(Location,+Loc:State,-Loc:Zip)            !Sort descending by zip in state
SORT(Location,'FirstField,-SecondField')     !Sort descending by city in name
```

SQL (use SQL code)

SQL(*sql expression*)

SQL Specifies a proper SQL expression to be used to affect the records in a VIEW.

sql expression A string expression containing valid SQL code.

SQL specifies a valid SQL statement which is applied to a FILTER, ORDER, JOIN, PROP:FILTER, PROP:ORDER, or PROP:JOINEXPRESSION surrounding it. Anything specified in the *sql expression* is treated as pure SQL code.

Example:

```
OrdFile      FILE,DRIVER('ODBC'),PRE(Ord)
PrimaryKey   KEY(Ord:OrdID),PRIMARY
Record       RECORD
OrdID        LONG
Customer     LONG
PurchaseDate DATE
Info         CSTRING(61)
            END
            END
```

```
MyView VIEW(AFile), FILTER('Ord:PurchaseDate = TODAY() AND ' & |
'SQL(A.Info LIKE "%Must deliver today%")'), ORDER('SQL(A.Customer)')
END
```

```
MyView{PROP:Filter} =
'SQL(A.Customer IN (SELECT CustID FROM Customers WHERE BadCustomer = 0))'
```

SQLCALLBACK (register or unregister a SQLCallbackInterface)

SQLCALLBACK(*entity*, *SQLCallbackInterface*, [*flag*])

SQLCALLBACK Register or unregister a SQLCallbackInterface.

entity The label of a FILE or VIEW.

SQLCallbackInterface

The label of the interface that implements the SQLCallbackInterface. The method of the SQLCallbackInterface (ExecutingCode) is called just before the SQL statement is passed to the SQL server for execution.

flag An integer constant, variable, EQUATE, or expression that indicates whether or not to unregister an interface associated with a FILE or VIEW. A value of one (1 or TRUE) unregisters the interface. If omitted, the interface is registered with the *entity*.

The **SQLCallback** method registers a callback interface with the specified entity. The methods of the registered interface are called whenever a file operation is done. Multiple interfaces can be registered with an entity.

To unregister an interface, set the *flag* to TRUE. Any registered interfaces must be unregistered before the object that implements the interface is removed.

Implementation: This function is only supported by the SQL drivers.

Example:

PROGRAM

```
MAP
END
INCLUDE ( 'FILECB.INC' ), ONCE
```

!Data file

```
People      FILE, DRIVER ( ' TOPSPEED' ), PRE ( PEO ), CREATE, BINDABLE, THREAD
KeyId       KEY ( PEO:Id ), NOCASE, OPT
KeyLastName KEY ( PEO:LastName ), DUP, NOCASE
Record      RECORD, PRE ( )
Id          LONG
FirstName   STRING ( 30 )
LastName    STRING ( 30 )
Gender      LONG
           END
           END
```

```

!DataFile
SQLFile FILE,DRIVER('MSSQL'),OWNER('(local),clarion,sa,;'),NAME('sptesttable')
Record RECORD
Id LONG
END
END

!LogFile
LogFile FILE,DRIVER('BASIC','/ALWAYSQUOTE=OFF /COMMA=1,1'),CREATE,NAME('logfile.txt')
Record RECORD
Operation STRING(200)
END
END

!FileCallback implementation
FCB CLASS,IMPLEMENTS(FileCallbackInterface),IMPLEMENTS(SQLCallbackInterface)
END

CODE
CALLBACK(People,FCB.FileCallbackInterface) !Register Interface
SQLCALLBACK(SQLFile,FCB.SQLCallbackInterface) !Register Interface
CREATE(Logfile) !Create log file
OPEN(Logfile) !Open log file
OPEN(People) !Open data file
SET(PEO:KeyId, PEO:KeyID) !Set and
LOOP !loop thru
NEXT(People) !data until
IF ERRORCODE() THEN BREAK END !end of file
END
CLOSE(People) !Close data file

SQLFile{PROP:SQL} = 'CREATE TABLE sptesttable (c INT)' !Create SQL table
OPEN(SQLFile) !Open SQL table
SQLFile.Id=5
ADD(SQLFile) !Update SQL table
SQLFile.Id=7
ADD(SQLFile) !Update SQL table
SQLFile.Id=8
ADD(SQLFile) !Update SQL table
CLOSE(SQLFile) !Close SQL table
CALLBACK(People,FCB.FileCallbackInterface, TRUE) !Unregister Interface
SQLCALLBACK(SQLFile,FCB.SQLCallbackInterface, TRUE) !Unregister Interface

```

!This method is called prior to each operation of the data file.

!The log file is updated

!with the file operation that is being executed.

FCB.FileCallBackInterface.FunctionCalled |

```
PROCEDURE(SIGNED opCode, *Params Parameters, *CSTRING ErrCode, *CSTRING ErrMsg)
```

p LIKE(Params)

CODE

```
p = Parameters
```

```
CASE opCode
```

```
OF DriverOp:ADD
```

```
  logFile.Operation = 'ADD(f)'
```

```
OF DriverOp:APPEND
```

```
  logFile.Operation = 'APPEND(f)'
```

```
OF DriverOp:CLOSE
```

```
  logFile.Operation = 'CLOSE(f)'
```

```
OF DriverOp:COPY
```

```
  logFile.Operation = 'COPY(f, ''&CLIP(Parameters.Text)&'')'
```

```
OF DriverOp:CREATE
```

```
  logFile.Operation = 'CREATE(f)'
```

```
OF DriverOp:DELETE
```

```
  logFile.Operation = 'DELETE(f)'
```

```
OF DriverOp:NEXT
```

```
  logFile.Operation = 'NEXT(f)'
```

```
OF DriverOp:OPEN
```

```
  logFile.Operation = 'OPEN(f, '&Parameters.openMode&' )'
```

```
OF DriverOp:PUT
```

```
  logFile.Operation = 'PUT(f)'
```

```
OF DriverOp:SETkeykey
```

```
  logFile.Operation = 'SET(k,k)'
```

```
END
```

```
ADD(logFile)
```

```
RETURN TRUE
```

!This method is called after each operation to the data file. This simply returns a TRU

!according to the rules of the FileCallBackInterface.

FCB.FileCallBackInterface.FunctionDone |

```
PROCEDURE(SIGNED opCode, Params Parameters, *CSTRING ErrCode, *CSTRING ErrMsg)
```

CODE

```
RETURN TRUE
```

!This method is called just before the SQL statement is passed to the SQL server for

!execution

FCB.SQLCallBackInterface.ExecutingCode |

```
PROCEDURE(CONST *CSTRING inStr, *BYTE Err, *CSTRING ErrCode, *CSTRING ErrMsg)
```

CODE

```
Err = FALSE
```

```
RETURN inStrPUT_re_write_record
```

SQRT (return square root)

SQRT(*expression*)

SQRT Returns square root.

expression A numeric constant, variable, or expression. If the value of the expression is less than zero, the return value is zero.

The **SQRT** procedure returns the square root of the *expression*. If X represents any positive real number, the square root of X is a number that, when multiplied by itself, produces a product equal to X .

Return Data Type: REAL

Example:

```
Length = SQRT(X^2 + Y^2) !Find the distance from 0,0 to x,y (pythagorean theorem)
```

START (return new execution thread)

START(*procedure* [, *stack*] [, *passed value*])

START	Begins a new execution thread.
<i>procedure</i>	The label of the first PROCEDURE to call on the new execution thread. This may not be an overloaded procedure.
<i>stack</i>	An integer constant or variable containing the size of the stack to allocate to the new execution thread. If omitted, the default stack is 10,000 bytes.
<i>passed value</i>	A string constant, variable, or expression containing the value to pass as a parameter to the <i>procedure</i> . There may be up to three <i>passed values</i> listed.

The **START** procedure begins a new execution thread, calling the *procedure* and returning the number assigned to the new thread. The returned thread number is used by procedures and procedures whose action may be performed on any execution thread (such as SETTARGET). The maximum number of simultaneously available execution threads in a single application is 64.

Code execution in the launching thread immediately continues with the next statement following the START and continues until an ACCEPT statement executes. Once the launching thread executes ACCEPT, the launched *procedure* begins executing its code in its new thread, retaining control until it executes an ACCEPT.

The *procedure* may be prototyped to receive up to three STRING parameters (passed by value) which may not be omitted. The values to pass to the *procedure* are listed as the *passed values* parameters to the START statement, and not in a parameter list attached to the *procedure* within the START statement. The procedure may not be an overloaded procedure.

The first execution thread in any program is the main program code, which is always numbered one (1). Therefore, the lowest value START can return is two (2), when the first START procedure is executed in a program. START may return zero (0), which indicates failure to open the thread. This can occur by running out of memory, or by starting a thread when the system is modal.

Return Data Type: SIGNED

Example:

```
MAP
NewProc1  PROCEDURE
NewProc2  PROCEDURE ( STRING )
NewProc3  PROCEDURE ( STRING , STRING )
NewProc4  PROCEDURE ( STRING , STRING , STRING )
END
```

```

MainWin APPLICATION('My Application'),SYSTEM,MAX,ICON('MyIcon.ICO'),STATUS |
,HVSCROLL,RESIZE
  MENUBAR
    MENU('&File'),USE(?FileMenu)
      ITEM('Selection &1...'),USE(?MenuSelection1)
      ITEM('Selection &2...'),USE(?MenuSelection2)
      ITEM('Selection &3...'),USE(?MenuSelection3)
      ITEM('Selection &4...'),USE(?MenuSelection4)
      ITEM('E&xit'),USE(?Exit)
    END
  END
END
SaveThread1 LONG           !Declare thread number save variables
SaveThread2 LONG
SaveThread3 LONG
SaveThread4 LONG
GroupName GROUP
F1 STRING(30)
F2 LONG
END

CODE
OPEN(MainWin)             !Open the APPLICATION
ACCEPT
CASE ACCEPTED()
OF ?MenuSelection1
  !Start thread with 35K stack
  SaveThread1 = START(NewProc1,35000)
OF ?MenuSelection2
  !Start thread, passing 1 parm
  SaveThread2 = START(NewProc2,35000,GroupName)
OF ?MenuSelection3
  !Start thread, passing 2 parms
  SaveThread3 = START(NewProc3,35000,'X','21')
OF ?MenuSelection4
  !Start a new thread
  SaveThread4 = START(NewProc4,35000,'X','21',GroupName)
OF ?Exit
  RETURN
END
END

NewProc2 PROCEDURE(MyGroup)
LocalGroup GROUP(GroupName) !Declare local group same as passed group
END

CODE
LocalGroup = MyGroup       !Get the passed data

```

See Also: ACCEPT, THREAD, SETTARGET, POST

STATUS (return file or window/report status)

```
STATUS( file      )
       | window |
```

STATUS	Returns the passed file, window or report status.
<i>file</i>	The label of a FILE statement.
<i>window</i>	The label of a WINDOW, APPLICATION, or REPORT structure, or valid built-in variables TARGET, PRINTER, or SYSTEM.

The **STATUS** procedure returns zero (0) if the *file* is not open, and the *file's access mode* if it is open. If the *access mode* is actually zero (Read Only / Any Access), 40h (Read Only / Deny None) is returned (see OPEN).

STATUS can also be used to check the state of a WINDOW or REPORT structure. If the passed WINDOW or REPORT exists and is opened, **STATUS** returns zero (0). If the passed WINDOW or REPORT exists but either has not been opened or has been closed, **STATUS** returns one (1). If the passed parameter is not a valid WINDOW or REPORT structure, **STATUS** returns two (2).

To simplify testing of results returned from calls to **STATUS**(WINDOW), the following declarations have been added to EQUATES.CLW:

```
WINDOW:OK           EQUATE(0)
WINDOW:NotOpened   EQUATE(1)
WINDOW:BadWindow   EQUATE(2)
```

Built-in variables TARGET, PRINTER and SYSTEM can be passed as parameters to calls to STATUS(WINDOW). The function always returns WINDOW:OK if the actual parameter is PRINTER or SYSTEM.

Return Data Type: LONG

Example:

```
IF STATUS(DataFile) % 16 = 0      !Opened Read-Only?
  RETURN                          !get out
ELSE                              !Otherwise
  EXECUTE DiskAction             !Write record to disk
  ADD(DataFile)
  PUT(DataFile)
  DELETE(DataFile)
  END
END

CASE STATUS(PassedLabel)         !Check parameter passed
OF WINDOW:OK                    !if window is good, and opened
  CYCLE                          !continue
OF WINDOW:NotOpened             !if window is good, but not opened
  OPEN(PassedLabel)             !open it
OF WINDOW:BadWindow             !bad window?
  MESSAGE('Invalid Window','Note:') !exit procedure
  RETURN
END
```

See Also: OPEN

STOP (suspend program execution)

STOP([*message*])

STOP Suspends program execution and displays a message window.

message An optional string expression (up to 64K) which displays in the error window.

STOP suspends program execution and displays a message window. It offers the user the option of continuing the program or exiting. When exiting, it closes all files and frees the allocated memory.

Example:

```
PswdScreen WINDOW
    STRING(' Please Enter the Password '),AT(5,5)
    ENTRY(@10),AT(20,5),USE(Password),PASSWORD
    END

CODE
OPEN(PswdScreen)           !Open the password screen
ACCEPT                     !and get user input
CASE ACCEPTED
OF ?Password)
  IF Password <> 'PayMe$moRe' !Correct password?
  STOP('Incorrect Password Entered -- Access Denied -- Retry?')
  X# += 1
  IF X# > 3                 !Let them try 3 times
  HALT(0,'Incorrect password') !then throw them out
  END
END
END
END
END
```

See Also:

HALT

STREAM (enable operating system buffering)

STREAM(*file*)

STREAM Disables automatic FILE flushing.

file The label of a FILE.

Some file systems flush the operating system's buffers on each disk write keeping the file "logically closed" (for example, the Clarion and TopSpeed file drivers do this by default). The **STREAM** statement disables this automatic flushing operation. A STREAM operation is terminated by closing the file, which automatically flushes the buffers, or by issuing a FLUSH statement.

STREAM and FLUSH are inherently single-user, batch process type of statements, although you can use them in networked environments. In some file systems, STREAM and FLUSH are simply ignored when the file is opened for shared access while in others they execute but it is possible to create a "deadlock" situation between workstations for multiple file updates. LOGOUT accomplishes a similar purpose in multi-user environments and is much safer.

Support for this statement is dependent upon the file system and is described in its file driver's documentation.

Example:

```

STREAM(History)            !Use DOS buffering
SET(Current)              !Set to top of current file
LOOP
  NEXT(Current)
  IF ERRORCODE( ) THEN BREAK END
  His:Record = Cur:Record
  ADD(History)
END
FLUSH(History)            !End streaming, flush buffers

```

See Also:

FLUSH

LOGOUT

BUFFER

STRPOS (return matching value position)

STRPOS(*first*, *second* [, *mode*])

STRPOS Returns the starting position of a substring based on all parameters passed.

first A string containing data to compare against the *second* parameter.

second A string containing a regular expression to match the *first* parameter.

mode An integer constant or equate which specifies if the comparison method is case sensitive. If FALSE (0), the method is case sensitive. If TRUE (1), the method is not case sensitive. The default value is case sensitive.

The **STRPOS** procedure returns the starting position where the first and second parameters match according to the comparison mode specified. The following mode value EQUATE, listed in EQUATES.CLW, is supported by STRPOS:

Match:Regular

A type of search where the string passed as a first parameter contains the regular expression passed as a second parameter. Repeated usage with the same regular expression value is optimized (to avoid re-compiling the expression).

Regular Expression Operators

Regular expressions are used to describe patterns in text. The following characters are regular expression operators (or metacharacters) used to increase the power and versatility of regular expressions.

^ Caret matches the beginning of the string. For example:

^@chapter

matches the “@chapter” at the beginning of a string.

\$ Dollar sign is similar to the caret, but it matches only at the end of a string. For example:

p\$

matches a record that ends with a p.

. Period matches any single character except a new line. For example:

.P

matches any single character followed by a P in a string. Using concatenation we can make regular expressions like `U.A', which matches any three-character sequence that begins with `U' and ends with `A'.

- [...]** This is called a character set. It matches any one of the characters that are enclosed in the square brackets. For example:
- `[MVX]`
- matches any one of the characters M, V, or X in a string. Ranges of characters are indicated by using a hyphen between the beginning and ending characters, and enclosing the whole thing in brackets. For example:
- `[0-9]`
- matches any digit. To match '-', write it as `---`, which is a range containing only '-'. You may also give '-' as the first or last character in the set. To match '^', put it anywhere except as the first character of a set. To match a `]`, make it the first character in the set. For example:
- `[]d^`
- matches either `]`, `d` or `^`.
- [^ ...]** This is a complemented character set. The first character after the [must be a ^. It matches any characters except those in the square brackets (or new line). For example:
- `[^0-9]`
- matches any character that is not a digit.
- |** Vertical bar is the alternation operator and it is used to specify alternatives. For example:
- `^P|[0-9]`
- matches any string that matches either ^P or [0-9]. This means it matches any string that contains a digit or starts with P. The alternation applies to the largest possible regular expressions on either side.
- {...}** Parentheses are used for grouping in regular expressions as in arithmetic. They can be used to concatenate regular expressions containing the alternation operator, |.

- * Asterisk means that the preceding regular expression is to be repeated as many times as possible to find a match. For example:

`ph*`

applies the * symbol to the preceding h and looks for matches to one p followed by any number of h's. This will also match just p if no h's are present. The * repeats the smallest possible preceding expression (use parentheses if you wish to repeat a larger expression). It finds as many repetitions as possible. For example:

`(c[ad][ad]*r x)`

matches a string of the form (car x), (cdr x), (cadr x), and so on.

- + Plus sign is similar to *, but the preceding expression must be matched at least once. This means that:

`wh+y`

would match “why” and “whhy” but not “wy,” whereas wh*y would match all three of these strings. This is a simpler way of writing the last * example:

`(c[ad]+r x)`

- ? Question mark is similar to *, but the preceding expression can be matched once or not at all. For example:

`fe?d`

will match fed and fd, but nothing else.

- \ Backslash is used to suppress the special meaning of a character when matching. For example:

`\$`

matches the character \$.

In regular expressions, the *, +, and ? operators have the highest precedence, followed by concatenation, and finally by |.

Return Data Type: LONG

Example:

```
ListHave1  STRING('IN,OH,KY,TN,PA')
ListHave2  STRING('WI,MN,IA,SD,ND')
StatesWanted STRING('NJ|NY|PA|DE')
CODE
X = STRPOS(ListHave1,StatesWanted,True)      !X = 13
Y = STRPOS(ListHave2,StatesWanted,True)      !Y = 0

X = STRPOS('Fireworks on the fourth', '{{4|Four}th', True) !X = 18
X = STRPOS('Fireworks on the fourth', '{{4|Four}th', False) !X = 0 Case sensitive
X = STRPOS('July 4th fireworks', '{{4|four}th', True)      !X = 6
```

See Also:

[MATCH](#)

SUB (return substring of string)

SUB(*string,position,length*)

SUB	Returns a portion of a string.
<i>string</i>	A string constant, variable or expression.
<i>position</i>	A integer constant, variable, or expression. If positive, it points to a character position relative to the beginning of the <i>string</i> . If negative, it points to the character position relative to the end of the <i>string</i> (i.e., a <i>position</i> value of -3 points to a position 3 characters from the end of the <i>string</i>).
<i>length</i>	A numeric constant, variable, or expression of number of characters to return.

The **SUB** procedure parses out a sub-string from a *string* by returning *length* characters from the *string*, starting at *position*.

The SUB procedure is similar to the "string slicing" operation on STRING, CSTRING, and PSTRING variables. SUB is less flexible and efficient than string slicing, but SUB is "safer" because it ensures that the operation does not overflow the bounds of the *string*.

"String slicing" is more flexible than SUB because it may be used on both the destination and source sides of an assignment statement, while the SUB procedure can only be used as the source. It is more efficient because it takes less memory than individual character assignments or the SUB procedure (however, no bounds checking occurs).

To take a "slice" of a string, the beginning and ending character numbers are separated by a colon (:) and placed in the implicit array dimension position within the square brackets ([]) of the string. The position numbers may be integer constants, variables, or expressions. If variables are used, there must be at least one blank space between the variable name and the colon separating the beginning and ending number (to prevent PREFIX confusion).

Return Data Type: **STRING**

Example:

```
!SUB('ABCDEFGHI',1,1) returns 'A'  
!SUB('ABCDEFGHI',-1,1) returns 'I'  
!SUB('ABCDEFGHI',4,3) returns 'DEF'  
!Get the file extension using SUB procedure  
Extension = SUB(fileName,INSTRING('.',fileName,1,1)+1,3)  
  
!The same operation using string slicing  
Extension = fileName[(INSTRING('.',fileName,1,1)+1) : (INSTRING('.',fileName,1,1)+3)]
```

See Also:

INSTRING

STRING

CSTRING

PSTRING

String Slicing

SUSPEND (suspend thread execution)

SUSPEND(*threadno*, *quietmode*)

SUSPEND Suspend thread execution.

threadno A numeric constant, variable, or expression that can be evaluated as a SIGNED integer.

quietmode A numeric constant, variable, or expression that can be evaluated as a SIGNED integer.

The **SUSPEND** function suspends a thread specified by the *threadno* parameter. If the *threadno* parameter is a number of an active thread, its execution is suspended and a suspending counter is incremented. Each additional **SUSPEND** statement issued to the same active thread will increment the suspending counter by one. Therefore, a thread that has been suspended with a given number of **SUSPEND** statements can only resume thread execution when an equal number of RESUME statements has been executed.

The *quietmode* parameter controls the built-in messaging support provided by **SUSPEND**. If the *quietmode* parameter is set to TRUE (1), SUSPEND will not display a message if it has detected that the thread it is about to suspend is not safe. If the *quietmode* parameter is set to FALSE (0), a message will be displayed if the impending thread suspension is not safe.

SUSPEND returns TRUE (1) if a thread has been suspended and returns FALSE (0) if the thread could not be suspended.

Care should be taken with MDI programs using **SUSPEND**, as improper use can cause program lockups. All MDI child windows have an MDI client window as a parent, and the MDI client window can send rather than post messages to its child windows.

For example, calling the inter-thread SendMessage modal function causes the calling thread (the MDI client window) to suspend activity until the called thread (the MDI Child window) returns from the call. If the called thread is suspended, we would have a program lockup.

Suspending a thread that is not already active is ignored.

Return Data Type: SIGNED

Example:

```
IF NOT SUSPEND(ThreadNumVariable)
  MESSAGE('Thread could not be suspended')
END
```

!suspend thread execution number stored in variable

See Also: RESUME

TAN (return tangent)

TAN(*radians*)

TAN	Returns tangent.
<i>radians</i>	A numeric constant, variable or expression for the angle in radians. p is a constant which represents the ratio of the circumference and radius of a circle. There are $2p$ radians (or 360 degrees) in a circle.

The **TAN** procedure returns the trigonometric tangent of an angle measured in *radians*. The tangent is the ratio of the angle's opposite side divided by its adjacent side.

Return Data Type: REAL

Example:

```

PI          EQUATE(3.1415926535898)      !The value of PI
Rad2Deg     EQUATE(57.295779513082)     !Number of degrees in a radian
Deg2Rad     EQUATE(0.0174532925199)     !Number of radians in a degree
CODE
Angle = 45 * Deg2Rad                    !Translate 45 degrees to Radians
TangentAngle = TAN(Angle)               !Get the tangent of 45 degree angle

```

See Also:

ATAN

SIN

ASIN

COS

ACOS

THREAD (return current execution thread)

THREAD()

The **THREAD** procedure returns the currently executing thread number. The returned thread number can be used by procedures and procedures whose action may be performed on any execution thread (such as SETTARGET).

The maximum number of simultaneously available execution threads in a single application is 64. The first execution thread in any program is the main program code, which is always thread number one (1). Therefore, THREAD always returns a value in the range of one (1) to sixty-four (64).

Return Data Type: SIGNED

Example:

```

MainWin APPLICATION('My Application'),SYSTEM,MAX,ICON('MyIcon.ICO'),STATUS |
,HVSCROLL,RESIZE
    MENUBAR
        MENU('&File'),USE(?FileMenu)
            ITEM('Selection &1...'),USE(?MenuSelection1)
            ITEM('Selection &2...'),USE(?MenuSelection2)
        END
    END
END

SaveThread LONG !Declare thread number save variable
SaveThread1 LONG !Declare thread number save variable
SaveThread2 LONG !Declare thread number save variable
CODE
SaveThread = THREAD() !Save thread number
OPEN(MainWin) !Open the APPLICATION
ACCEPT !Handle Global events
CASE ACCEPTED()
OF ?MenuSelection1
SaveThread1 = START(NewProc1) !Start a new thread
OF ?MenuSelection2
SaveThread2 = START(NewProc2) !Start a new thread
OF ?Exit
RETURN
END
END

```

See Also: START

THREADLOCKED (returns current execution thread locked state)

THREADLOCKED()

The **THREADLOCKED** procedure returns the current execution thread's locked/unlocked state. It returns zero (0) if the thread is unlocked and one (1) if the thread is locked.

Return Data Type: SIGNED

Example:

```
X# = THREADLOCKED()           !Returns 1
UNLOCKTHREAD                 !Unlock the thread
X# = THREADLOCKED()           !Returns 0
MyLibraryCodeWithMessageLoop !Call the code that has its own message loop
LOCKTHREAD                   !Re-lock the thread
```

See Also:

ACCEPT

LOCKTHREAD

UNLOCKTHREAD

TIE(associate a string value to an ASTRING)

TIE([*stringtoken*] [,*index*] [,*value*])

TIE	Associates a string value to an ASTRING.
<i>stringtoken</i>	A string constant or the label of a previously declared ASTRING token.
<i>index</i>	A numeric constant or variable defining the index value of an ASTRING. The index value may be any LONG value greater than zero.
<i>value</i>	A numeric constant or variable. The value may be any LONG value greater than zero.

The **TIE** statement associates a *value* to a particular *stringtoken* and *index*. A unique index may be allocated at runtime by either omitting the index or setting it to zero. If the index is allocated at runtime, TIE will return the generated index number.

Return Data Type: LONG

Example:

```

PROGRAM
MAP
END

INCLUDE('EQUATES.CLW')
COLOR EQUATE(1)
SIDES EQUATE(2)

AS1 ASTRING

CODE
AS1 = 'SHAPE' !Declare ASTRING string token
TIE('SHAPE',COLOR,COLOR:RED) !Associate COLOR:RED to COLOR for ASTRING SHAPE
TIE(AS1,SIDES,3) !Associate 3 to SIDES for ASTRING SHAPE
IF TIED('SHAPE',SIDES) = 3 AND | !If the value associated to SIDES = 3
TIED(AS1,COLOR) = COLOR:RED !and the value associated with COLOR=COLOR:RED
MESSAGE('Shape is a red triangle')
END

```

See Also:

TIED

UNTIE

TIED(retrieves a value associated with an ASTRING)

TIED(*stringtoken*, *index*)

TIED	Retrieves a string value that was previously associated to an ASTRING token.
<i>stringtoken</i>	A string constant or the label of a previously declared ASTRING token.
<i>index</i>	A numeric constant or variable defining the index value of an ASTRING. The index value may be any LONG value greater than zero.

The **TIED** statement retrieves the value that was previously associated to a particular ASTRING *stringtoken* and *index*. If a value is not TIED then 0 will be returned.

Return Data Type: LONG

Example:

```

PROGRAM
MAP
END

INCLUDE('EQUATES.CLW')
COLOR EQUATE(1)
SIDES EQUATE(2)

AS1 ASTRING

CODE
AS1 = 'SHAPE'
TIE('SHAPE',COLOR,COLOR:RED)
TIE(AS1,SIDES,3)
IF TIED('SHAPE',SIDES) = 3
AND TIED(AS1,COLOR) = COLOR:RED
MESSAGE('Shape is a red triangle')
END

```

See Also:

TIE

UNTIE

TODAY (return system date)

TODAY()

The **TODAY** procedure returns the operating system date as a standard date. The range of possible dates is from January 1, 1801 (standard date 4) to December 31, 2099 (standard date 109,211).

Return Data Type: LONG

Example:

```
OrderDate = TODAY() !Set the order date to system date
```

See Also:

Standard Date

DAY

MONTH

YEAR

SETTODAY

DATE

TYPE (write string to screen)

TYPE(*string*)

TYPE Writes a *string* to the current window or report.

string A string constant, variable, or expression.

TYPE writes a *string* to the current window or report. The *string* appears on the window or report at the current cursor position, if there's room, if not, it appears on the next line. The font used is the current font for the window or report. The SHOW statement may be used to position the cursor before output from TYPE.

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
    !window controls
    END
CODE
OPEN(MDIChild)
DISPLAY
TYPE(Cus:Notes)       !Type the notes field
```

UNHIDE (show hidden control)

UNHIDE([*first control*] [, *last control*])

UNHIDE Displays previously hidden controls.

first control Field number or field equate label of a control, or the first control in a range of controls. If omitted, defaults to zero (0).

last control Field number or field equate label of the last control in a range of controls.

The **UNHIDE** statement reactivates a control or range of controls, that were hidden by the HIDE statement. Once un-hidden, the control is again visible on screen.

Example:

```
CODE
OPEN(Screen)
HIDE(?Control2)           !Control2 is hidden
IF Ctl:Password = 'Supervisor'
  UNHIDE(?Control2)       !Unhide Control2
END
```

See Also:

HIDE

ENABLE

DISABLE

UNLOAD (remove a CALLED DLL from memory)

UNLOAD(*file*)

UNLOAD Unloads a Windows standard .DLL previously loaded by CALL.

file A string constant, variable, or expression containing the name (including extension) of the .DLL to unload. This may include a full path.

The **UNLOAD** procedure unloads a .DLL *file* left loaded by the CALL procedure.

Example:

```
Win1 WINDOW
    BUTTON('Load DLL'),USE(?DLLButton)
END

CODE
OPEN(Win1)
ACCEPT
CASE EVENT()
OF EVENT:Accepted
    IF ACCEPTED() = ?DLLButton
        !Call procedure in CUSTOM.DLL and leave DLL resident:
        X# = CALL('CUSTOM.DLL','EntryPoint',1)
        IF X# THEN STOP(X#) END !Check for successful execution
    END
OF EVENT:CloseWindow
    UNLOAD('CUSTOM.DLL') !Unload the CUSTOM.DLL
END
END
!Process
```

See Also:

CALL

UNLOCK (unlock a locked data file)

UNLOCK(*file*)

UNLOCK Unlocks a previously locked data file.

file The label of a FILE declaration.

The **UNLOCK** statement unlocks a previously LOCKed data file. It will not unlock a file locked by another user. If the *file* is not locked, or is locked by another user, UNLOCK is ignored. UNLOCK posts no errors.

Example:

```
LOOP                    !Loop to avoid "deadly embrace"
  LOCK(Master,1)        !Lock the master file, try for 1 second
  IF ERRORCODE() = 32 !If someone else has it
    CYCLE                !try again
  END
  LOCK(Detail,1)        !Lock the detail file, try for 1 second
  IF ERRORCODE() = 32 !If someone else has it
    UNLOCK(Master)      !unlock the locked file
    CYCLE                !try again
  END
  BREAK                 !Break loop when both files are locked
END
```

See Also:

LOCK

UNLOCKTHREAD (unlock the current execution thread)

UNLOCKTHREAD

The **UNLOCKTHREAD** statement allows a Clarion program to call 3rd-party code or API procedures that contain their own message loop (like Clarion's ACCEPT loop).

Normally, ACCEPT loops in a Clarion program execute in turn (even in 32-bit programs) so that problems do not arise due to simultaneous access to data. Thread-switching only occurs at an ACCEPT statement, and thus only one thread can execute Clarion code at a time. However, if the currently executing thread calls an external procedure (including API functions) that (for example) opens a window and processes messages until the window closes, then other threads must be allowed to execute (co-operatively) to process their own messages. You do this by calling UNLOCKTHREAD before the external procedure, then LOCKTHREAD after it returns.

Because UNLOCKTHREAD may allow other threads to pre-empt the currently executing thread, it is important that you make **NO** calls to the Clarion runtime library between UNLOCKTHREAD and its corresponding LOCKTHREAD. This means you must not call any Clarion language procedure. You also must not perform any operations involving a STRING, CSTRING, PSTRING, DECIMAL, or PDECIMAL data types. The one exception is that you may pass a STRING, CSTRING, or PSTRING variable as a RAW parameter to an external (non-Clarion) procedure. **Failure to observe this restriction may result in data on another thread becoming corrupted, or other generally unpredictable misbehavior.**

The THREADLOCKED() procedure determines whether the thread has been unlocked or not.

Example:

```
UNLOCKTHREAD          !Unlock the thread
MyLibraryCodeWithMessageLoop  !Call the code that has its own message loop
LOCKTHREAD           !Re-lock the thread
```

See Also:

ACCEPT

LOCKTHREAD

THREADLOCKED

UNQUOTE (remove string special characters)

UNQUOTE(*value*)

UNQUOTE Contracts the *value*'s string data, "singling up" doubled single quotes ("), and all doubled un-paired left angle brackets (<<) and left curly braces ({}).

value A string constant or variable containing the properties to parse.

The **UNQUOTE** procedure returns the string contained in the *value* 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).

Return Data Type: **STRING**

Example:

```
stringvar1 STRING('<250>')
stringvar2 STRING('<<display text>')
stringvar3 STRING(30)
```

CODE

```
stringvar3 = 'label{{PROP:text} = value'
```

```
MESSAGE(UNQUOTE(stringvar1)) !returns accented u
MESSAGE(UNQUOTE(stringvar2)) !returns '<display text>'
MESSAGE(UNQUOTE(stringvar3)) !returns ' label{PROP:text} = value '
```

See Also:

QUOTE

UNREGISTER (unregister event handler)

UNREGISTER([*event*] [, *handler*] [, *object*] [,*window*] [,*control*])

UNREGISTER Unregisters an event handling procedure.

<i>event</i>	An integer constant, variable, expression, or EQUATE containing an event number. A value in the range 400h to 0FFFh is a User-defined event. If omitted, all <i>events</i> are unregistered.
<i>handler</i>	A LONG variable, or expression containing the return value from ADDRESS for the PROCEDURE to handle the <i>event</i> . If omitted, all <i>handlers</i> are unregistered.
<i>object</i>	A LONG integer constant, variable, or expression containing any 32-bit unique value to identify the specific <i>handler</i> . This is generally the return value of ADDRESS(SELF) when the <i>handler</i> is a CLASS method.
<i>window</i>	The label of the WINDOW or REPORT whose <i>event</i> to handle. If omitted, the current target WINDOW or REPORT is assumed.
<i>control</i>	An integer constant, EQUATE, variable, or expression containing the field number of the specific control whose <i>event</i> to handle. If omitted, the <i>event</i> is handled for every control on the <i>window</i> .

UNREGISTER prevents a previously REGISTERed event *handler* PROCEDURE from being called to handle its *event*.

Example:

```
WindowResizeClass.Kill PROCEDURE
CODE
UNREGISTER(EVENT: Sized, ADDRESS( SELF.TakeResize ), ADDRESS( SELF ) )
!Other code follows

WindowResizeClass.TakeResize PROCEDURE
ReturnValue BYTE
CODE
ReturnValue = Level:Benign
RETURN(ReturnValue)
```

See Also:

REGISTER

ACCEPT

EVENT

UNTIE(disassociate a string value from an ASTRING)

UNTIE(*[stringtoken],[index]*)

UNTIE	Removes a value previously TIED for an ASTRING.
<i>stringtoken</i>	The label of a previously declared ASTRING token. If omitted, all previously declared ASTRINGs are used.
<i>index</i>	A numeric constant or variable defining the index value of a previously declared ASTRING. The index value may be any LONG value greater than zero. If omitted, all indexes are used.

The **UNTIE** statement disassociates a string value that was previously associated to a particular ASTRING *stringtoken* and *index*.

Example:

```

PROGRAM
MAP
END

INCLUDE('EQUATES.CLW')
COLOR EQUATE(1)
SIDES EQUATE(2)

AS1 ASTRING

CODE
AS1 = 'SHAPE' !Declare ASTRING string token
TIE('SHAPE',COLOR,COLOR:RED) !Associate COLOR:RED to COLOR for ASTRING SHAPE
TIE(AS1,SIDES,3) !Associate 3 to SIDES for ASTRING SHAPE
(some code here)
UNTIE(AS1, SIDES)

```

See Also:

TIE

TIED

UPDATE (write from screen to USE variables)

UPDATE([*first control*] [,*last control*])

UPDATE Writes the contents of a control to its USE variable.

first control Field number or field equate label of a control, or the first control in a range of controls.

last control Field number or field equate label of the last control in a range of controls.

UPDATE writes the contents of a screen control to its USE variable. This takes the value displayed on screen and places it in the variable specified by the control's USE attribute.

USE variables are updated automatically by ACCEPT as each control is accepted. However, certain events (such as an ALERTed key press) do not automatically update USE variables. This is the purpose of the UPDATE statement.

UPDATE
 Updates all controls on the screen.

UPDATE(*first control*)
 Updates a specific USE variable from its associated screen control.

UPDATE(*first control*,*last control*)
 Updates the USE variables of an inclusive range of screen controls.

Example:

```
UPDATE(?)           !Update the currently selected control
UPDATE             !Update all controls on the screen
UPDATE(?Address)   !Update the address control
UPDATE(3,7)        !Update controls 3 through 7
UPDATE(?Name,?Zip) !Update controls from name through zip
UPDATE(?City,?City+2) !Update city and 2 controls following
```

See Also:

Field Equate Labels

DISPLAY

CHANGE

UPPER (return upper case)

UPPER(*string*)

UPPER Returns all upper case string.

string A string constant, variable, or expression for the *string* to be converted.

The **UPPER** procedure returns a string with all letters converted to upper case.

Return Data Type: **STRING**

Example:

```
!UPPER('abc') returns 'ABC'
```

```
Name = UPPER(Name) !Make the name upper case
```

See Also:

LOWER

ISUPPER

ISLOWER

VAL (return ASCII value)

VAL(*character*)

VAL Returns ASCII code.

character A one-byte string containing an ANSI character.

The **VAL** procedure returns the ASCII code of a *character*.

Return Data Type: LONG

Example:

```
!VAL('A') returns 65
!VAL('z') returns 122
```

```
CharVal = VAL(StrChar) !Get the ASCII value of the string character
```

See Also:

CHR

WATCH (automatic concurrency check)

WATCH(*entity*)

WATCH Arms automatic optimistic concurrency checking.

entity The label of a FILE or VIEW declaration.

The **WATCH** statement arms automatic optimistic concurrency checking by the file driver for a following GET, REGET, NEXT, or PREVIOUS statement in a multi-user environment. The WATCH terminates when the WATCHed record is PUT back to the *entity*, or another GET, NEXT, PREVIOUS, or REGET statement executes on the same *entity* without first executing another WATCH statement.

Generally, the file driver retains a copy of the retrieved record on the GET, NEXT, PREVIOUS, or REGET when it successfully gets the record. When the retrieved record is PUT to the *file*, the record on disk is compared to the original record retrieved. Error 89 (Record Changed By Another Station) is posted by the PUT statement if the record has been changed by another user.

Example:

```

SET(Itm:InvoiceKey)           !Start at beginning of Items file
LOOP                          !Process all records
  WATCH(Items)               !Arm concurrency check
  NEXT(Items)                 !Get a record
  IF ERRORCODE() THEN BREAK.
  DO ItemProcess              !process the item
  PUT(Items)                  !and put it back
  IF ERRORCODE() = RecordChangedErr !If changed by another station
    PREVIOUS(Items)          !Setup to re-process the changed record
  ELSE
    STOP(ERROR())           !Stop on any other error
  END
END
END

```

See Also:

NEXT

PREVIOUS

GET

REGET

HOLD

WHAT (return field from group)

WHAT(*group*, *number*)

WHAT Returns a specified field from a *group* structure.

group The label of a GROUP, RECORD, CLASS, or QUEUE declaration.

number An integer expression specifying the ordinal position of a field in the *group*.

The **WHAT** statement returns the *number* specified field from a *group* structure. Generally, this would be assigned to an ANY variable.

If field with an ordinal position is equal to the passed second parameter, and is defined as an ANY type or has the DIM attribute, the returned value can be used only in "reference equality" (&=) operations. Any attempt to access these field types by returned reference will cause a runtime error.

Return Data Type: ANY

Example:

```
MyGroup  GROUP
F1       LONG(3)      !Field number 1
F2       SHORT        !Field number 2
F3       STRING(30)   !Field number 3
InGroup  GROUP        !Field number 4
F1       LONG         !Field number 5
F2       SHORT(2)     !Field number 6
F3       STRING(30),DIM(2) !Field number 7
F4       ANY
        END
        END
```

CurrentField ANY

```
CODE
CurrentField &= WHAT(MyGroup,1)      !Returns contents of MyGroup.F1 (3)

CurrentField &= WHAT(MyGroup,6)      !Returns contents of MyGroup.Ingroup.F2 (2)

CurrentField &= WHAT(MyGroup,7)      !INVALID - posts a runtime error
```

See Also:

ANY, WHERE, WHO, ISSTRING, GROUP

RECORD, CLASS, QUEUE

WHERE (return field position in group)

WHERE(*group*, *field*)

WHERE Returns a *field's* ordinal position within a GROUP, RECORD, CLASS, or QUEUE structure.

group The label of a GROUP, RECORD, CLASS, or QUEUE declaration.

field The label of a field in the *group* declaration.

The **WHERE** statement returns the ordinal position of a specified *field* in a *group* structure.

Return Data Type: SIGNED

Example:

```
MyGroup  GROUP
F1       LONG           !Field number 1
F2       SHORT          !Field number 2
F3       STRING(30)     !Field number 3
InGroup  GROUP          !Field number 4
F1       LONG           !Field number 5
F2       SHORT          !Field number 6
F3       STRING(30)     !Field number 7
        END
        END
CurrentField LONG
```

CODE

```
CurrentField = WHERE(MyGroup,MyGroup.F1)           !WHERE returns 1
CurrentField = WHERE(MyGroup,MyGroup.Ingroup.F2)  !WHERE returns 6
CurrentField = WHERE(MyGroup.Ingroup,MyGroup.Ingroup.F2) !WHERE returns 2
```

See Also:

WHAT

WHO

ISSTRING

GROUP

RECORD

CLASS

QUEUE

WHO (return field name from group)

WHO(*group*, *number*)

WHO Returns a string containing the name of a specified field from a *group* structure.

group The label of a GROUP, RECORD, CLASS, or QUEUE declaration with the BINDABLE attribute.

number An integer expression specifying the ordinal position of a field in the *group*.

The **WHO** statement returns a string containing the name of the *number* specified field from a *group* structure.

Return Data Type: **STRING**

Example:

```
MyGroup GROUP
F1      LONG          !Field number 1
F2      SHORT        !Field number 2
F3      STRING(30)   !Field number 3
InGroup GROUP       !Field number 4
F1      LONG          !Field number 5
F2      SHORT        !Field number 6
F3      STRING(30)   !Field number 7
      END
      END
```

```
CurrentField STRING(30)
```

```
CODE
```

```
CurrentField = WHO(MyGroup,1)      !Returns "MyGroup.F1"
```

```
CurrentField = WHO(MyGroup,6)      !Returns "MyGroup.Ingroup.F2"
```

See Also:

WHAT

WHERE

GROUP

RECORD

CLASS

QUEUE

BINDABLE

YEAR (return year of date)

YEAR(*date*)

YEAR	Returns the year.
<i>date</i>	A numeric constant, variable, expression, or the label of a string variable declared with a date picture, containing a standard date. A variable declared with a date picture is automatically converted to a standard date intermediate value.

The **YEAR** procedure returns a four digit number for the year of a standard *date* (1801 to 9999).

Return Data Type: **LONG**

Example:

```
IF YEAR>LastOrd) < YEAR(TODAY())    !If last order date not from this year
  DO StartNewYear                    !start new year to date totals
END
```

See Also:

Standard Date

DAY

MONTH

TODAY

DATE

YIELD (allow event processing)

YIELD

YIELD temporarily gives control to Windows to allow other concurrently executing Windows applications to process events they need to handle (except those events that would post messages back to the program containing the YIELD statement, or events that would change focus to the other application).

YIELD is used to ensure that long batch processing in a Clarion application does not completely "lock out" other applications from completing their tasks. This is known as "cooperative multi-tasking" and ensures that your Windows programs peacefully co-exist with any other Windows applications.

Within your Clarion application, YIELD only allows control to pass to EVENT:Timer events in other execution threads. This allows you to code a "background" procedure in its own execution thread using the TIMER attribute to perform some long batch processing without requiring the user to wait until the task is complete before continuing with other work in the application. This is an industry-standard Windows method of doing background processing within an application.

The example code on the next page demonstrates both approaches to performing batch processing: making the user wait for the process to complete, and processing in the background. Only the WaitForProcess procedure requires the YIELD statement, because it takes full control of the program. Background processing using EVENT:Timer does not need a YIELD statement, since the ACCEPT loop automatically performs cooperative multi-tasking with other Windows applications.

Example:

```

StartProcess PROCEDURE
Win WINDOW('Choose a Batch Process'),MDI
  BUTTON('Full Control'),USE(?FullControl)
  BUTTON('Background'),USE(?Background)
  BUTTON('Close'),USE(?Close)
END
CODE
OPEN(Win)
ACCEPT
CASE ACCEPTED()
OF ?FullControl
  DISABLE(FIRSTFIELD(),LASTFIELD())      !Disable all buttons
  WaitForProcess                          !and call the batch process procedure
  ENABLE(FIRSTFIELD(),LASTFIELD())        !Enable buttons when batch is complete
OF ?Background
  START(BackgroundProcess)                !Start new execution thread for the process

```

```

    OF ?Close
      BREAK
    END
  END
END

WaitForProcess PROCEDURE          !Full control Batch process
CODE
SETCURSOR(CURSOR:Wait)          !Alert user to batch in progress
SET(File)                        !Set up a batch process
LOOP
  NEXT(File)
  IF ERRORCODE() THEN BREAK END
  !Perform some batch processing code
  YIELD                          !Yield to other applications and EVENT:Timer
END
SETCURSOR                        !Restore mouse cursor

BackgroundProcess PROCEDURE      !Background processing batch process
Win WINDOW('Batch Processing...'),TIMER(1),MDI
  BUTTON('Cancel'),STD(STD:Close)
  END
CODE
OPEN(Win)
SET(File)                        !Set up a batch process
ACCEPT
CASE EVENT()
OF EVENT:CloseWindow
  BREAK
OF EVENT:Timer                  !Process records whenever the timer allows it
  LOOP 3 TIMES
    NEXT(File)
    IF ERRORCODE()
      POST(EVENT:CloseWindow)
      BREAK
    END
    !Perform some batch processing code
  END
END
END
END

```

See Also:

ACCEPT

TIMER

Appendix A - DDE, OLE, and .OCX

Dynamic Data Exchange

DDE Overview

Dynamic Data Exchange (DDE) is a very powerful Windows tool that allows a user to access data from another separately executing Windows application. This allows the user to work with the data in its native format (in the originating application), while ensuring that the application in which the data is used always has the most current values.

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 client.

To be a DDE server, a Clarion application must:

- Open at least one window, since all DDE servers must be associated with a window (and its ACCEPT loop).
- Register with Windows as a DDE server, using the DDESERVER procedure.
- Provide the requested data to the client, using the DDEWRITE statement.
- When DDE is no longer required, terminate the link by using the DDECLOSE statement. You can also allow it to terminate when the user closes the server application or the window that started the link.

To be a DDE client, a Clarion application must:

- Open at least one window, since all DDE events must be processed with a window's ACCEPT loop.
- Open a link to a DDE server as its client, using the DDECLIENT procedure.
- 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. You can also allow it to terminate automatically when the user closes the client window or application.

The DDE procedures are prototyped in the DDE.CLW file, which you must INCLUDE in your program's MAP structure. 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.

DDE Events

The DDE process is governed by several field-independent events specific to DDE. These events are posted to the ACCEPT loop of the window that opened the link between applications, either as a server or client.

The following events are posted only to a Clarion server application:

- EVENT:DDErequest
A client has requested a data item.
- EVENT:DDEadvise
A client has requested continuous updates of a data item.
- EVENT:DDEexecute
A client has executed a DDEEXECUTE statement.
- EVENT:DDEpoke
A client has sent unsolicited data

The following events are posted only to a Clarion client application:

- EVENT:DDEdata
A server has supplied an updated data item.
- EVENT:DDEclosed
A server has terminated the DDE link.

When one of these DDE events occur there are several procedures that tell you what posted the event:

- DDECHANNEL() returns the handle of the DDE server or client.
- DDEITEM() returns the item or command string passed to the server by the DDEREAD or DDEEXECUTE statements.
- DDEVALUE returns a string containing the data sent to the Clarion DDE server by the DDEPOKE statement, or the command to execute from a DDEEXECUTE statement.
- DDEAPP() returns the name of the application.
- DDETOPIC() returns the name of the topic.

When a Clarion program creates a DDE server, external clients can link to this server and request data. Each data request is accompanied by a string (in some specific format which the server program knows) indicating the required data item. If the Clarion server already knows the value for a given item, it supplies it to the client automatically without generating any events. If it doesn't, an EVENT:DDErequest or EVENT:DDEadvise event is posted to the server window's ACCEPT loop.

When a Clarion program creates a DDE client, it can link to external servers which can provide data. When the server first provides the value for a given item, it supplies it to the client automatically without generating any events. If the client has established a "hot" link with the server, an EVENT:DDEdata event is posted to the client window's ACCEPT loop whenever the server posts a new value for the data item.

DDE Procedures

DDEACKNOWLEDGE (send acknowledgement from DDE server)

DDEACKNOWLEDGE(*response*)

DDEACKNOWLEDGE

Sends acknowledgement of the current DDEPOKE or DDEEXECUTE statement sent to the DDE server.

response An integer constant, variable, or expression containing zero (0) or one (1) to indicate negative or positive acknowledgement.

The **DDEACKNOWLEDGE** procedure allows a DDE server program to immediately acknowledge unsolicited data sent from DDEPOKE, or commands sent from DDEEXECUTE. This allows the client application to immediately continue its processing. Although a CYCLE statement after EVENT:DDEpoke or EVENT:DDEexecute also signals positive acknowledgement to the client, DDEACKNOWLEDGE allows you to send negative acknowledgement.

Example:

```

!The client application's code contains:
WinOne WINDOW,AT(0,0,160,400)
    END
SomeServer LONG
DDEChannel LONG
CODE
    OPEN(WinOne)
    DDEChannel = DDECLIENT('MyServer','System') !Open a channel to MyServer app
    DDEEXECUTE(DDEChannel,['ShowList'])      !Tell it to do something

```

```

!The server application's code contains:
WinOne WINDOW,AT(0,0,160,400)
    END
DDEChannel LONG
CODE
    OPEN(WinOne)
    DDEChannel = DDESERVER('MyServer','System') !Open channel
    ACCEPT
    CASE EVENT()
    OF EVENT:DDEExecute
    CASE DDEVALUE()
    OF 'ShowList'
        DDEACKNOWLEDGE(1)
        DO ShowList
    ELSE
        DDEACKNOWLEDGE(0)
    END
    END
END
END

```

See Also:

DDEPOKE

DDEEXECUTE

DDEAPP (return server application)**DDEAPP()**

The **DDEAPP** procedure returns a string containing the application name in the DDE channel that has just posted a DDE event. This is usually the same as the first parameter to the DDESERVER or DDECLIENT procedure when the DDE channel is established.

Return Data Type: **STRING**

Example:

```
ClientApp STRING(20)
WinOne WINDOW,AT(0,0,160,400)
        STRING(@S20),AT(5,5,90,20),USE(ClientApp)
        END

TimeServer LONG
DateServer LONG
FormatTime STRING(5)
FormatDate STRING(8)

CODE
OPEN(WinOne)
TimeServer = DDESERVER('SomeApp','Time')      !Open as server
DateServer = DDESERVER('SomeApp','Date')      !Open as server
ACCEPT
CASE EVENT()
OF EVENT:DDErequest
CASE DDECHANNEL()
OF TimeServer
    ClientApp = DDEAPP()                        !Get client's name
    DISPLAY                                     !and display on screen
    FormatTime = FORMAT(CLOCK(),@T1)
    DDEWRITE(TimeServer,DDE>manual,'Time',FormatTime)
OF DateServer
    ClientApp = DDEAPP()                        !Get client's name
    DISPLAY                                     !and display on screen
    FormatDate = FORMAT(TODAY(),@D1)
    DDEWRITE(DateServer,DDE>manual,'Date',FormatDate)
    END
END
END
END
```

See Also: DDECLIENT, DDESERVER

DDECHANNEL (return DDE channel number)

DDECHANNEL()

The **DDECHANNEL** procedure returns a LONG integer containing the channel number of the DDE client or server application that has just posted a DDE event. This is the same value returned by the **DDESERVER** or **DDECLIENT** procedure when the DDE channel is established.

Return Data Type: LONG

Example:

```
WinOne WINDOW,AT(0,0,160,400)
    END
TimeServer LONG
DateServer LONG
FormatTime STRING(5)
FormatDate STRING(8)
CODE
OPEN(WinOne)
TimeServer = DDESERVER('SomeApp','Time')    !Open as server
DateServer = DDESERVER('SomeApp','Date')    !Open as server
ACCEPT
CASE EVENT()
OF EVENT:DDErequest
    CASE DDECHANNEL()                        !Check which channel
    OF TimeServer
        FormatTime = FORMAT(CLOCK(),@T1)
        DDEWRITE(TimeServer,DDE>manual,'Time',FormatTime)
    OF DateServer
        FormatDate = FORMAT(TODAY(),@D1)
        DDEWRITE(DateServer,DDE>manual,'Date',FormatDate)
    END
END
END
END
```

See Also:

DDECLIENT

DDESERVER

DDECLIENT (return DDE client channel)

DDECLIENT([*application*] [, *topic*])

DDECLIENT Returns a new DDE client channel number.

application A string constant or variable containing the name of the server application to link to. Usually, this is the name of the application. If omitted, the first DDE server application available is used.

topic A string constant or variable containing the name of the application-specific topic. If omitted, the first topic available in the *application* is used.

The **DDECLIENT** procedure returns a new DDE client channel number for the *application* and *topic*. If the *application* is not currently executing, DDECLIENT returns zero (0).

Typically, when opening a DDE channel as the client, the *application* is the name of the server application. The *topic* is a string that the *application* has either registered with Windows as a valid *topic* for the *application*, or represents some value that tells the *application* what data to provide. You can use the DDEQUERY procedure to determine the *applications* and *topics* currently registered with Windows.

Return Data Type: LONG

Example:

```

DDEReadVal REAL
WinOne WINDOW,AT(0,0,160,400)
    ENTRY(@s20),USE(DDEReadVal)
    END
ExcelServer LONG
CODE
OPEN(WinOne)

!Open as client to Excel spreadsheet:
ExcelServer = DDECLIENT('Excel','MySheet.XLS')
IF NOT ExcelServer           !If the server is not running
    MESSAGE('Please start Excel') !alert the user to start it
    RETURN                   !and try again
END
DDEREAD(ExcelServer,DDE:auto,'R5C5',DDEReadVal)
ACCEPT
CASE EVENT()
OF EVENT:DDEdata           !As changed data comes from Excel
    PassedData(DDEReadVal) !process the new data
END
END

```

See Also: DDEQUERY, DDEWRITE, DDESERVER

DDECLOSE (terminate DDE server link)

DDECLOSE(*channel*)

DDECLOSE Closes an open DDE channel.

channel The label of the LONG integer variable containing the channel number--the value returned by the DDESERVER or DDECLIENT procedure.

The **DDECLOSE** procedure allows a DDE client program to terminate the specified *channel*. A *channel* is automatically terminated when the window which opened the *channel* is closed.

Errors Posted: 601 Invalid DDE Channel
 602 DDE Channel Not Open
 605 Time Out

Example:

```
WinOne WINDOW,AT(0,0,160,400)
      END
SomeServer LONG
CODE
OPEN(WinOne)
SomeServer = DDECLIENT('SomeApp','MyTopic') !Open as client
ACCEPT
END
DDECLOSE(SomeServer)
```

See Also:

DDECLIENT

DDESERVER

Example:

```
!The client application's code contains:
WinOne WINDOW,AT(0,0,160,400)
    END
SomeServer LONG
DDEChannel LONG
CODE
    OPEN(WinOne)

!Open a channel to Windows Program Manager:
DDEChannel = DDECLIENT('PROGMAN','PROGMAN')

!Create a new program group:
DDEEXECUTE(DDEChannel,['CreateGroup(Clarion Applications)'])

!Display it:
DDEEXECUTE(DDEChannel,['ShowGroup(1)'])

!Create new item in the group using second icon in progman.exe
DDEEXECUTE(DDEChannel,['AddItem(MYAPP.EXE,My Program,PROGMAN.EXE,2)'])
```

See Also:

DDEACKNOWLEDGE

DDEVALUE

DDEITEM (return server item)**DDEITEM()**

The **DDEITEM** procedure returns a string containing the name of the item for the current DDE event. This is the item requested by a DDEREAD, or the data item supplied by DDEPOKE.

Return Data Type: STRING

Example:

```
WinOne WINDOW,AT(0,0,160,400)
      END
```

```
Server      LONG
FormatTime  STRING(5)
FormatDate  STRING(8)
```

```
CODE
OPEN(WinOne)
Server = DDESERVER('SomeApp','Clock') !Open as server for my topic
ACCEPT
CASE EVENT()
OF EVENT:DDErequest
CASE DDEITEM()
OF 'Time'
FormatTime = FORMAT(CLOCK(),@T1)
DDEWRITE(Server,DDE:manual,'Time',FormatTime)
OF 'Date'
FormatDate = FORMAT(TODAY(),@D1)
DDEWRITE(Server,DDE:manual,'Date',FormatDate)
END
OF EVENT:DDEadvise
CASE DDEITEM()
OF 'Time'
FormatTime = FORMAT(CLOCK(),@T1)
DDEWRITE(Server,1,'Time',FormatTime)
OF 'Date'
FormatDate = FORMAT(TODAY(),@D1)
DDEWRITE(Server,60,'Date',FormatDate)
END
END
END
```

See Also: DDEREAD, DDEEXECUTE, DDEPOKE

DDEPOKE (send unsolicited data to DDE server)

DDEPOKE(*channel*, *item*, *value*)

DDEPOKE Sends unsolicited data through an open DDE client channel to a DDE server.

channel A LONG integer constant or variable containing the client channel--the value returned by the DDECLIENT procedure.

item A string constant or variable containing the application-specific item to receive the unsolicited data.

value A string constant or variable containing the data to place in the *item*.

The **DDEPOKE** procedure allows a DDE client program to communicate unsolicited data to the server. The *item* and *value* parameters must be in a format the server application can recognize and act on. The server does not need to be a Clarion program.

A Clarion DDE server can use the DDEITEM() and DDEVALUE() procedures to determine what the client has sent. The CYCLE statement after an EVENT:DDEpoke signals positive acknowledgement to the client that sent the unsolicited data. DDEACKNOWLEDGE can send either positive or negative acknowledgement.

Errors Posted: 601 Invalid DDE Channel
 602 DDE Channel Not Open
 604 DDEPOKE Failed
 605 Time Out

Events Generated: EVENT:DDEpoke A client has sent unsolicited data

Example:

```

WinOne WINDOW,AT(0,0,160,400)
    END
DDEChannel LONG
    CODE
    OPEN(WinOne)
    DDEChannel = DDECLIENT('Excel','System')           !Open channel to Excel
    DDEEXECUTE(DDEChannel,['NEW(1)'])                  !Create a new spreadsheet
    DDEEXECUTE(DDEChannel,['Save.As("DDE_CHART.XLS)']) !Save it as DDE_CHART.XLS
    DDECLOSE(DDEChannel)                               !Close conversation
    DDEChannel = DDECLIENT('Excel','DDE_CHART.XLS')   !Open channel to new chart
    DDEPOKE(DDEChannel,'R1C2','Widgets')              !Send it data
    DDEPOKE(DDEChannel,'R1C3','Gadgets')
    DDEPOKE(DDEChannel,'R2C1','East')
    DDEPOKE(DDEChannel,'R3C1','West')
    DDEPOKE(DDEChannel,'R2C2','450')
    DDEPOKE(DDEChannel,'R3C2','275')
    DDEPOKE(DDEChannel,'R2C3','340')
    DDEPOKE(DDEChannel,'R3C3','390')
    DDEEXECUTE(DDEChannel,['SELECT("R1C1:R3C2)'])     !Highlight the data
    DDEEXECUTE(DDEChannel,['NEW(2,2)'])               !and create a new chart
    !Send some more commands here to format the chart and work with it
    DDECLOSE(DDEChannel)                               !Close channel when done

```

See Also: DDEACKNOWLEDGE, DDEITEM, DDEVALUE

DDEQUERY (return registered DDE servers)

DDEQUERY([*application*][, *topic*])

DDEQUERY Returns currently executing DDE servers.

application A string constant or variable containing the name of the application to query. For most applications, this is the name of the application. If omitted, all registered *applications* registered with the specified *topic* are returned.

topic A string constant or variable containing the name of the application-specific topic to query. If omitted, all *topics* for the *application* are returned.

The **DDEQUERY** procedure returns a string containing the names of the currently available DDE server *applications* and their *topics*.

If the *topic* parameter is omitted, all *topics* for the specified *application* are returned. If the *application* parameter is omitted, all registered *applications* registered with the specified *topic* are returned. If both parameters are omitted, DDEQUERY returns all currently available DDE servers.

The format of the data in the return string is *application:topic* and it can contain multiple *application* and *topic* pairs delimited by commas (for example, 'Excel:MySheet.XLS,ClarionApp:DataFile.DAT').

Return Data Type: STRING

Example:

```
!This example code does not handle DDEADVISE
WinOne WINDOW,AT(0,0,160,400)
    END
SomeServer LONG
ServerString STRING(200)
CODE
OPEN(WinOne)
LOOP
    ServerString = DDEQUERY()                !Return all registered servers
    IF NOT INSTRING('SomeApp:MyTopic',ServerString,1,1)
        MESSAGE('Open SomeApp, Please')
    ELSE
        BREAK
    END
END
SomeServer = DDECLIENT('SomeApp','MyTopic') !Open as client
ACCEPT
END
DDECLCLOSE(SomeServer)
```

DDEREAD (get data from DDE server)

DDEREAD(*channel*, *mode*, *item* [, *variable*])

DDEREAD	Gets data from a previously opened DDE client channel.
<i>channel</i>	A LONG integer constant or variable containing the client channel--the value returned by the DDECLIENT procedure.
<i>mode</i>	An EQUATE defining the type of data link: DDE:auto, DDE>manual, or DDE:remove (defined in EQUATES.CLW).
<i>item</i>	A string constant or variable containing the application-specific name of the data item to retrieve.
<i>variable</i>	The name of the variable to receive the retrieved data. If omitted and <i>mode</i> is DDE:remove, all links to the <i>item</i> are canceled.

The **DDEREAD** procedure allows a DDE client program to read data from the *channel* into the *variable*. The type of update is determined by the *mode* parameter. The *item* parameter supplies some string value to the server application that tells it what specific data item is being requested. The format and structure of the *item* string is dependent upon the server application.

If the *mode* is DDE:auto, the *variable* is continually updated by the server (a "hot" link). An EVENT:DDEdata is generated each time the *variable* is updated by the server.

If the *mode* is DDE>manual, the *variable* is updated once and no event is generated. Another DDEREAD request must be sent to the server to check for any changed value (a "cold" link).

If the *mode* is DDE:remove, a previous "hot" link to the *variable* is terminated. If the *mode* is DDE:remove and *variable* is omitted, all previous "hot" links to the *item* are terminated, no matter what *variables* were linked. This means the client must send another DDEREAD request to the server to check for any changed value.

Errors Posted: 601 Invalid DDE Channel
 602 DDE Channel Not Open
 605 Time Out

Events Generated: These events are posted to the client application:

EVENT:DDEdata
 A server has supplied an updated data item for a hot link.

EVENT:DDEclosed
 A server has terminated the DDE link.

Example:

```
WinOne WINDOW,AT(0,0,160,400)
      END
```

```
ExcelServer LONG(0)
DDEReadVal REAL
```

```
CODE
```

```
OPEN(WinOne)
```

```
!Open as client to Excel spreadsheet
```

```
ExcelServer = DDECLIENT('Excel','MySheet.XLS')
```

```
IF NOT ExcelServer           !If the server is not running
```

```
  MESSAGE('Please start Excel') !alert the user to start it
```

```
  CLOSE(WinOne)
```

```
  RETURN
```

```
END
```

```
!Request continual update from server:
```

```
DDEREAD(ExcelServer,DDE:auto,'R5C5',DDEReadVal)
```

```
ACCEPT
```

```
  CASE EVENT()
```

```
    OF EVENT:DDEdata           !As changed data comes from Excel
```

```
      PassedData(DDEReadVal)   ! call proc to process the new data
```

```
    END
```

```
END
```

See Also:

DDEQUERY

DDEWRITE

DDESERVER

DDESERVER (return DDE server channel)

DDESERVER([*application*] [, *topic*])

DDESERVER Returns a new DDE server channel number.

application A string constant or variable containing the name of the application. Usually, this is the name of the application. If omitted, the filename of the application (without extension) is used.

topic A string constant or variable containing the name of the application-specific topic. If omitted, the *application* will respond to any data request.

The **DDESERVER** procedure returns a new DDE server channel number for the *application* and *topic*. The channel number specifies a *topic* for which the *application* will provide data. This allows a single Clarion *application* to register as a DDE server for multiple *topics*.

Return Data Type: LONG

Example:

```

DDERetVal  STRING(20)
WinOne     WINDOW,AT(0,0,160,400)
           ENTRY(@s20),USE(DDERetVal)
           END
MyServer   LONG

CODE
OPEN(WinOne)
MyServer = DDESERVER('MyApp','DataEntered')  !Open as server
ACCEPT
CASE EVENT()
OF EVENT:DDerequest           !As server for data requested once
  DDEWRITE(MyServer,DDE:manual,'DataEntered',DDERetVal) !Provide data once
OF EVENT:DDEadvise           !As server for constant update request
  DDEWRITE(MyServer,15,'DataEntered',DDERetVal)
                               !Check for change every 15 seconds
                               !and provide data whenever changed

END
END

```

See Also:

DDECLIENT

DDEWRITE

DDETOPIC (return server topic)**DDETOPIC()**

The **DDETOPIC** procedure returns a string containing the topic name for the DDE channel that has just posted a DDE event.

Return Data Type: **STRING**

Example:

```
WinOne WINDOW,AT(0,0,160,400)
    END

TimeServer LONG
DateServer LONG
FormatTime STRING(5)
FormatDate STRING(8)

CODE
OPEN(WinOne)
TimeServer = DDESERVER('SomeApp')      !Open as server
ACCEPT
CASE EVENT()
OF EVENT:DDErequest
CASE DDETOPIC()                          !Get requested topic
OF 'Time'
    FormatTime = FORMAT(CLOCK(),@T1)
    DDEWRITE(TimeServer,DDE:manual,'Time',FormatTime)
OF 'Date'
    FormatDate = FORMAT(TODAY(),@D1)
    DDEWRITE(DateServer,DDE:manual,'Date',FormatDate)
END
END
END
```

See Also:

DDERead

DDECLIENT

DDESERVER

DDEVALUE (return data value sent to server)**DDEVALUE()**

The **DDEVALUE** procedure returns a string containing the data sent to a Clarion DDE server by the DDEPOKE statement, or the command to execute from a DDEEXECUTE statement.

Return Data Type: **STRING**

Example:

```
WinOne WINDOW,AT(0,0,160,400)
      END
```

```
TimeServer LONG
```

```
TimeStamp FILE,DRIVER(ASCII),PRE(Tim)
Record     RECORD
FormatTime STRING(5)
FormatDate STRING(8)
Message    STRING(50)
      END
      END
```

```
CODE
OPEN(WinOne)
TimeServer = DDESERVER('TimeStamp')      !Open as server
ACCEPT
CASE EVENT()
OF EVENT:DDEpoke
  OPEN(TimeStamp)
  Tim:FormatTime = FORMAT(CLOCK(),@T1)
  Tim:FormatDate = FORMAT(TODAY(),@D1)
  Tim:Message = DDEVALUE()                !Get data
  ADD(TimeStamp)
  CLOSE(TimeStamp)
  CYCLE                                   !Ensure acknowledgement
END
END
```

See Also:

DDEPOKE

DDEEXECUTE

DDEWRITE (provide data to DDE client)

DDEWRITE(*channel*, *mode*, *item* [, *variable*])

DDEWRITE	Provide data to an open DDE server channel.
<i>channel</i>	A LONG integer constant or variable containing the server channel--the value returned by the DDESERVER procedure.
<i>mode</i>	An integer constant or variable containing the interval (in seconds) to poll for changes to the <i>variable</i> , or an EQUATE defining the type of data link: DDE:auto, DDE:manual, or DDE:remove (defined in EQUATES.CLW).
<i>item</i>	A string constant or variable containing the application-specific name of the data item to provide.
<i>variable</i>	The name of the variable providing the data. If omitted and <i>mode</i> is DDE:remove, all links to the <i>item</i> are canceled.

The **DDEWRITE** procedure allows a DDE server program to provide the *variable's* data to the client. The *item* parameter supplies a string value that identifies the specific data item being provided. The format and structure of the *item* string is dependent upon the *server* application. The type of update performed is determined by the *mode* parameter.

If the *mode* is DDE:auto, the client program receives the current value of the variable and the internal libraries continue to provide that value whenever the client (or any other client) asks for it again. If the client requested a "hot" link, any changes to the *variable* should be tracked by the Clarion program so it can issue a new DDEWRITE statement to update the client with the new value.

If the *mode* is DDE:manual, the *variable* is updated only once. If the client requested a "hot" link, any changes to the *variable* should be tracked by the Clarion program so it can issue a new DDEWRITE statement to update the client with the new value. PROP:DDETimeOut can be used to set or get the time out value for the DDE connection (default is five seconds).

If the *mode* is a positive integer, the internal libraries check the value of the *variable* whenever the specified number of seconds has passed. If the value has changed, the client is automatically updated with the new value by the internal libraries (without the need for any further Clarion code). This can incur significant overhead, depending upon the data, and so should be used only when necessary.

If the *mode* is DDE:remove, any previous "hot" link to the *variable* is terminated. If the *mode* is DDE:remove and *variable* is omitted, all previous "hot" links to the *item* are terminated, no matter what *variables* were linked. This means the client must send another DDEREAD request to the server to check for any changed value.

Errors Posted: 601 Invalid DDE Channel
 602 DDE Channel Not Open
 605 Time Out

Events Generated:
 EVENT:DDErequest
 A client has requested a data item (a "cold" link).
 EVENT:DDEadvise
 A client has requested continuous updates of a data item (a "hot" link).

Example:

```

DDERetVal  STRING(20)
WinOne     WINDOW,AT(0,0,160,400)
           ENTRY(@s20),USE(DDERetVal)
           END
MyServer   LONG
CODE
OPEN(WinOne)
MyServer = DDESERVER('MyApp','DataEntered')  !Open as server
ACCEPT
CASE EVENT()
OF EVENT:DDErequest           !As server for data requested once
  DDEWRITE(MyServer,DDE:manual,'DataEntered',DDERetVal)  !Provide data once
OF EVENT:DDEadvise           !As server for constant update request
  DDEWRITE(MyServer,15,'DataEntered',DDERetVal)
                               !Check for change every 15 seconds
                               !and provide data whenever changed

END
END
  
```

See Also:

DDEQUERY

DDERead

DDESERVER

Object Linking and Embedding

OLE Overview

Object Linking and Embedding (OLE) allows "objects" from one application to be linked or embedded into a "document" (data structure) of another application. The application that creates and maintains the object is an OLE Server application, while the application that contains the object is an OLE Controller application (sometimes referred to as an OLE Client application). OLE "objects" are data structures appropriate to the OLE Server application (such as a chart from a spreadsheet, or an image from a paint or drawing application). The object is placed in a "container window" in the Controller application. In Clarion for Windows, "container windows" are OLE controls.

Clarion's implementation of OLE allows a Clarion for Windows application to serve as an OLE Controller application, linking or embedding objects from any OLE Server application. Clarion also supports OLE Automation, which gives an OLE Controller application dynamic control of the OLE Server application, using the OLE Server's macro language.

Object Linking

Object Linking generally means the OLE Controller application stores "pointer" information to the object, whether that object is the entire data structure (like a spreadsheet file) or a component of the data structure (like a range of cells in a spreadsheet). When an Object is linked into the OLE Controller application the OLE Controller application contains only the information necessary to reference the linked data. This can be stored in either a BLOB or an OLE Compound Storage file.

Object Embedding

Object Embedding generally means the OLE Controller application stores the entire object, independently of the OLE Server application. An Object embedded into the OLE Controller application does not exist as a separate data file which the OLE Server application may access. The OLE Controller application completely contains the active object, which can be stored in either a BLOB or an OLE Compound Storage file.

Maintaining the OLE Object

Any object in the OLE Controller application, whether linked or embedded, is maintained by the OLE Server application which created it, not the OLE Controller application. This means that when the user wants to change the object, the OLE Controller application activates the OLE Server application to make the changes. There are two ways to activate an OLE Server: "in-place activation" and "open-mode."

In-place activation

In-place activation means the user seems to stay in the OLE Controller application, but the OLE Server's menus and toolbar merge into the OLE Controller's menus and toolbar and the OLE Server is the currently executing application. The Object being edited has a "hash-mark" border to indicate that it is in edit-mode.

If the OLE Server application has one or more toolbars then the toolbars will appear either as pop-up toolbars or as toolbars attached to one of the edges of the frame, or a combination of both. This can appear to "push down" the controls on your window, so take care designing your window.

Open-mode activation

Open-mode activation means the user is switched into the OLE Server application executing in a separate window. The Object being edited is in the Server application and ready to edit, while the Object in the OLE Controller application has "hash-marks" completely covering it to indicate a separate window is editing the object.

OLE Container Properties

Contents:

Attribute Properties
 Undeclared Properties
 Example Program
 Interface Properties
 Clarion OLE/OCX library and object hierarchies

There are a number of properties associated with an OLE container control that deal only with OLE objects (not .OCX controls).

Attribute Properties

PROP:Create The CREATE attribute (blank if none). (WRITE ONLY)
 PROP:Open The OPEN attribute (blank if none). (WRITE ONLY)
 PROP:Document
 The DOCUMENT attribute (blank if none). (WRITE ONLY)
 PROP:Link The LINK attribute (blank if none). (WRITE ONLY)
 PROP:Clip The CLIP attribute. A toggle attribute. Assigning a null string (") or zero turns it off, and '1' or 1 turns it on. (WRITE ONLY)
 PROP:Stretch The STRETCH attribute. A toggle attribute. Assigning a null string (") or zero turns it off, and '1' or 1 turns it on. (WRITE ONLY)
 PROP:Autosize The AUTOSIZE attribute. A toggle attribute. Assigning a null string (") or zero turns it off, and '1' or 1 turns it on. (WRITE ONLY)
 PROP:Zoom The ZOOM attribute. A toggle attribute. Assigning a null string (") or zero turns it off, and '1' or 1 turns it on. (WRITE ONLY)
 PROP:Compatibility
 The COMPATIBILITY attribute (blank if none). (WRITE ONLY)

Undeclared Properties

PROP:Blob Convert an object to and from a blob. (READ/WRITE)
 PROP:SaveAs Saves the object to an OLE Compound Storage file. (WRITE ONLY)
 The syntax for placing the object in the file is '*filename\!component*'. For example:

```
?controlx{PROP:SaveAs} = 'myfile\!objectx'
```

PROP:DoVerb	Executes an OLE doverb command from the following set of commands (WRITE ONLY):
	<p>DOVERB:Primary (0) Calls the object's primary action. The object, not the container, determines this action. If the object supports in-place activation, the primary verb usually activates the object in-place.</p> <p>DOVERB:Show (-1) Tells the object to show itself for editing or viewing. Called to display newly inserted objects for initial editing and to show link sources. This is usually an alias for some other object-defined action.</p> <p>DOVERB:Open (-2) Tells the object to open itself for editing in a separate window from its container (this includes objects that support in-place activation). If the object does not support in-place activation, this has the same action as DOVERB:Show.</p> <p>DOVERB:Hide (-3) Tells the object to remove its user interface. This applies only to objects activated in-place.</p> <p>DOVERB:UIActivate (-4) Activates the object in place, along with its full set of user-interface tools, including menus, toolbars, and its name in the title bar of the container window.</p> <p>DOVERB:InPlaceActivate (-5) Activates the object in-place without displaying the tools (menus and toolbars) that end-users need to change the behavior or appearance of the object.</p> <p>DOVERB:DiscardUndoState (-6) Tells the object to discard any undo state that it may be maintaining without deactivating the object.</p> <p>DOVERB:Properties (-7) Invokes the modal system property browser for the object to allow the user to set its properties.</p>
PROP:Deactivate	Deactivates an in-place active OLE object. (READ/WRITE/EXECUTE)
PROP:Update	Tells the OLE object to update itself. (READ/WRITE/EXECUTE)
PROP:CanPaste	Can you paste the object in the clipboard? (READ ONLY)
PROP:Paste	Pastes an object from the clipboard to an OLE container control. (READ/WRITE/EXECUTE)
PROP:CanPasteLink	Can the object in the clipboard be pasted as a link? (READ ONLY)

PROP:PasteLink	Pastes and links an object from the clipboard to an OLE container control. (READ/WRITE/EXECUTE)
PROP:Copy	Copies an object in an OLE container control to the clipboard. (READ/WRITE/EXECUTE)
PROP:ReportException	Report OLE exceptions (for debug). (WRITE ONLY)
PROP:OLE	Is there an OCX or OLE object in the container? (READ ONLY)
PROP:Language	The number for the language used for OLE Automation or OCX Method. The number for US English is 0409H, and other language numbers can be computed from the data in the WINNT.H file in the MS Windows SDK. (READ/WRITE)

Example Program:

```

PROGRAM
MAP
  INCLUDE('OCX.CLW')
SelectOleServer  PROCEDURE(OleQ PickQ),STRING
END
  INCLUDE 'XL.CLW'          !Constants that Excel uses
  INCLUDE 'ERRORS.CLW'     !Include errorcode constants
SaveLinks  FILE,DRIVER('TopSpeed'),PRE(SAV),CREATE
Object     BLOB
Record     RECORD
LinkType   STRING(1)       !F = File, B = BLOB
LinkFile   STRING(64)      !OLE Compound Storage file name and object
          END
          END

i         LONG             !Loop counters
j         LONG
ResultQ   QUEUE           !Queue to hold return from OLEDIRECTORY
Name      CSTRING(64)
CLSID     CSTRING(64)
ProgID    CSTRING(64)
          END
MainWin   WINDOW('OLE Demo'),AT(,,350,200),STATUS(-1,-1),SYSTEM,GRAY,RESIZE,MAX,TIMER(1)
          MENUBAR
            MENU('&File')
              ITEM('e&xit'),USE(?exit)
            END
            MENU('&Objects')
              ITEM('Create Object'),USE(?CreateObject)
              ITEM('Paste Object'),USE(?PasteObject)
              ITEM('PasteLink Object'),USE(?PasteLinkObject)
              ITEM('Save Object to BLOB'),USE(?SaveObjectBlob),DISABLE
              ITEM('Save Object to OLE File'),USE(?SaveObjectFile),DISABLE
              ITEM('Retrieve Saved Object'),USE(?GetObject),DISABLE
            END
            MENU('&Activate')
              ITEM('&Spreadsheets'),USE(?ActiveExcel)
              ITEM('&Any OLE Object'),USE(?ActiveOLE),DISABLE
            END
          END
OLE,AT(5,10,160,100),COLOR(0808000H),USE(?ExcelObject)
          MENUBAR
            MENU('&Clarion App')
              ITEM('&Deactivate Excel'),USE(?DeactExcel)
            END
          END
END

```

```

END
OLE,AT(170,10,160,100),USE(?AnyOLEObject),AUTOSIZE
MENUBAR
MENU('&Clarion App')
ITEM('&Deactivate Object'),USE(?DeactOLE)
END
END
END
END

CODE
OPEN(SaveLinks)
IF ERRORCODE()                                !Check for error on Open
IF ERRORCODE() = NoFileErr                    !if the file doesn't exist
CREATE(SaveLinks)                             !then create it
IF ERRORCODE() THEN HALT(,ERROR()) END
OPEN(SaveLinks)                               !then open it for use
IF ERRORCODE() THEN HALT(,ERROR()) END
ELSE
HALT(,ERROR())
END
END
OPEN(MainWin)
?ExcelObject{PROP:Create} = 'Excel.Sheet.5' !Create an Excel spreadsheet object
DO BuildSheetData                             !populate it with some random data
IF RECORDS(SaveLinks)                        !Check for existing saved record
SET(SaveLinks)                               !and get it
NEXT(SaveLinks)
POST(EVENT:Accepted,?GetObject)             !and display it
DO MenuEnable
ELSE
ADD(SaveLinks)                               !or add blank record
END
IF ERRORCODE() THEN HALT(,ERROR()) END
ACCEPT
CASE EVENT()
OF EVENT:CloseWindow
?ExcelObject{PROP:Deactivate}                !Deactivate the OLE Server applications
?AnyOLEObject{PROP:Deactivate}
OF EVENT:Timer
IF CLIPBOARD()
IF ?AnyOLEObject{PROP:CanPaste}              !Can Paste object from the clipboard?
IF ?PasteObject{PROP:Disable}
ENABLE(?PasteObject)
END
ELSIF NOT ?PasteObject{PROP:Disable}
DISABLE(?PasteObject)
END
END

```

```

IF ?AnyOLEObject{PROP:CanPasteLink}           !Can PasteLink object from clipboard?
  IF ?PasteLinkObject{PROP:Disable}
    ENABLE(?PasteLinkObject)
  END
ELSIF NOT ?PasteLinkObject{PROP:Disable}
  DISABLE(?PasteLinkObject)
END
END
OF EVENT:Accepted
CASE FIELD()
OF ?Exit
  POST(EVENT:CloseWindow)
OF ?CreateObject
  OLEDIRECTORY(ResultQ,0)                       !Get a list of installed OLE Servers
  ?AnyOLEObject{PROP:Create} = SelectOleServer(ResultQ) !Let the user pick one
  ?AnyOLEObject{PROP:DoVerb} = 0                 !Activate OLE Server in its default mode
  DO MenuEnable
OF ?PasteObject
  ?AnyOLEObject{PROP:Paste}                     !Paste the object
  SETCLIPBOARD('Paste Completed')              !Assign non-object text to clipboard
  DO MenuEnable
OF ?PasteLinkObject
  ?AnyOLEObject{PROP:PasteLink}                 !PasteLink the object
  SETCLIPBOARD('PasteLink Completed')          !Assign non-object text to clipboard
  DO MenuEnable
OF ?SaveObjectBlob                             !Save object to BLOB
  SAV:Object{PROP:Handle} = ?AnyOLEObject{PROP:Blob}
  SAV:LinkType = 'B'
  PUT(SaveLinks)
  IF ERRORCODE() THEN STOP(ERROR()) END
OF ?SaveObjectFile                             !Save to OLE Compound Storage file
  ?AnyOLEObject{PROP:SaveAs} = 'TEST1.OLE\!Object'
  SAV:LinkFile = 'TEST1.OLE\!Object'
  SAV:LinkType = 'F'
  PUT(SaveLinks)
  IF ERRORCODE() THEN STOP(ERROR()) END
OF ?GetObject
  IF SAV:LinkType = 'F'                         !Saved to OLE Compound Storage file?
    ?AnyOLEObject{PROP:Open} = SAV:LinkFile
  ELSIF SAV:LinkType = 'B'                       !Saved to BLOB?
    ?AnyOLEObject{PROP:Blob} = SAV:Object{PROP:Handle}
  END
  DISPLAY
OF ?ActiveExcel
  ?ExcelObject{PROP:DoVerb} = 0                 !In-place activate Excel
OF ?ActiveOLE
  ?AnyOLEObject{PROP:DoVerb} = 0                 !Activate OLE Server in its default mode
OF ?DeactExcel

```

```

    ?ExcelObject{PROP:Deactivate}           !Return to the Clarion application
  OF ?DeactOLE
    ?AnyOLEObject{PROP:Deactivate}         !Return to the Clarion application
  END !CASE FIELD()
  END !CASE EVENT0()
  END !ACCEPT

BuildSheetData  ROUTINE                    !Use OLE Automation to build spreadsheet
?ExcelObject{PROP:ReportException} = TRUE !Excel will report any errors
?ExcelObject{'Application.Calculation'} = xlManual !turn off auto recalc
LOOP i = 1 TO 3                             !Fill Sheet with some values
  LOOP j = 1 TO 3
    ?ExcelObject{'Cells(' & i & ',' & j & ').Value'} = Random(100,900)
  END
  ?ExcelObject{'Cells(4,' & i & ').Value'} = 'Sum'
  ?ExcelObject{'Cells(5,' & i & ').FormulaR1C1'} = '=SUM(R[-4]C:R[-2]C)'
  ?ExcelObject{'Cells(6,' & i & ').Value'} = 'Average'
  ?ExcelObject{'Cells(7,' & i & ').FormulaR1C1'} = '=AVERAGE(R[-6]C:R[-4]C)'
  END
!turn auto recalc back on
?ExcelObject{'Application.Calculation'} = xlAutomatic
DISPLAY

MenuEnable  ROUTINE                        !Enable menu items
  ENABLE(?ActiveOLE)
  ENABLE(?SaveObjectBlob,?GetObject)

SelectOleServer PROCEDURE(OleQ PickQ)
window WINDOW('Choose OLE Server'),AT(,,122,159),CENTER,SYSTEM,GRAY
  LIST,AT(11,8,100,120),USE(?List),HVSCROLL, |
  FORMAT('146L~Name~@s64@135L~CLSID~@s64@20L~ProgID~@s64@'),FROM(PickQ)
  BUTTON('Select'),AT(42,134),USE(?Select)
  END

CODE
OPEN(window)
SELECT(?List,1)
ACCEPT
CASE ACCEPTED()
OF ?Select
  GET(PickQ,CHOICE(?List))
  IF ERRORCODE() THEN STOP(ERROR()) END
  POST(EVENT:CloseWindow)
END
END
RETURN(PickQ.ProgID)

```

Interface Properties

PROP:Object Gets the Dispatch interface for the object. (READ ONLY)

In VB the toolbar and tree control use the image-list control to show icons in the tree control and on the buttons on the toolbar. To associate an image control with a toolbar, use the following code:

```
?toolbar{'ImageList'} = ?imagelist{prop:object}
```

PROP>SelectInterface

Selects the interface to use with the object. (WRITE ONLY)

```
?x{PROP>SelectInterface} = 'x.y'
?x{'z(1)'} = 1
?x{'z(2)'} = 2
```

has the same meaning as

```
?x{'x.y.z(1)'} = 1
?x{'x.y.z(2)'} = 2
```

PROP:AddRef Increments the reference count for an interface. (WRITE ONLY)

PROP:Release Decrements the reference count for an interface. (WRITE ONLY)

Clarion OLE/OCX library and object hierarchies:

At the time of the design and implementation of the Clarion OLE library, the lack of access to secondary objects created by a primary object (example from excel: ExcelUse{'Application.Charts.Add'}), was not considered a problem as there were other ways of accessing the object(ExcelUse {'Application.Charts(Chart1).ChartWizard(' &?ex{'Range(A5:C5)'}&','&xl3DPie&',7,1,0,0,2,,,')})

At the time there was only one known instance where this was not the case. This is probably still true today as the OLE standard states that an object implement a collection, must also implement a method for accessing the objects by indexing.

Due to the special case mentioned above, when an object was created by one control and passed onto an other object as a parameter, a method which would be more or less transparent to the user, was implemented.

Calling a method which returns an IDispatchInterface is converted into a special representation (a '' followed by a number of digits). This special representation is recognized in a couple of places in the OLE library.

The place that you will find most useful is, when the special representation is in the place where an interface could occur in the property syntax it will replace any previous interface in the access of the properties or methods of the object. For example:

```
x=y{'charts.add()'}
y{x&'p(7)'}
```

where y is an ole object and x is a cstring. This is an example of a method returning an interface and later this interface is used to access a method p with the parameter 7.

In this context a further complication arises from the reference counting used in OLE. Which means that if the object are used more than once it must have it's reference count increased before use.

```
x=y{'charts.add()'}
y{PROP:AddRef}=x
y{x&'p(7)' }
y{x&'p(7)' } ;last use of x
```

OLEDIRECTORY (get list of installed OLE/OCX)

OLEDIRECTORY(*list* , *flag* [, *bits*])

OLEDIRECTORY

Gets a list of all installed OLE servers or OCX controls.

<i>list</i>	The label of the QUEUE structure to receive the list.
<i>flag</i>	An integer constant or variable that determines whether to get a list of OLE servers (<i>flag</i> = 0) or OCX controls (<i>flag</i> = 1).
<i>bits</i>	An integer constant or variable that determines whether to get a list of 16-bit or 32-bit OCX controls. If one (1), it returns 16-bit OCX controls. If two (2), it returns 32-bit OCX controls. If three (3), it returns both 16-bit and 32-bit OCX controls. If omitted or zero, 16-bit programs return 16-bit OCX controls and 32-bit programs return 32-bit OCX controls.

Note: With 16-bit support deprecated in this release, the 32-bit mode should be used exclusively.

OLEDIRECTORY gets a list of all installed OLE servers or OCX controls and places it in the *list* QUEUE. The *list* QUEUE must be declared with the same structure as the OleQ QUEUE declaration in EQUATES.CLW:

```
OleQ  QUEUE,TYPE
Name  CSTRING(64)  !Name of the OLE Server application
CLSID  CSTRING(64)  !Unique identifier for the operating system
ProgID CSTRING(64)  !Registry name, such as: Excel.Sheet.5
      END
```

Example:

```
ResultQ QUEUE(OleQ).      !Declare ResultQ the same as OleQ QUEUE in EQUATES.CLW
CODE
OLEDIRECTORY(ResultQ,0)  !Get list of installed OLE Servers & put it in ResultQ
                        !then let the user pick one:
?OleControl{PROP:Create} = SelectOleServer(ResultQ)

!User's OLE Server choice procedure:
SelectOleServer PROCEDURE(OleQ PickQ)
window WINDOW('Choose OLE Server'),AT(, ,122,159),CENTER,SYSTEM,GRAY
    LIST,AT(11,8,100,120),USE(?List),HVSCROLL, |
    FORMAT(' 146L~Name~@s64@135L~CLSID~@s64@20L~ProgID~@s64@'),FROM(PickQ)
    BUTTON('Select'),AT(42,134),USE(?Select)
      END
CODE
OPEN(window)
SELECT(?List,1)
```

```
ACCEPT
CASE ACCEPTED()
OF ?Select
  GET(PickQ,CHOICE(?List))
  IF ERRORCODE() THEN STOP(ERROR()) END
  POST(EVENT:CloseWindow)
END
END
RETURN(PickQ.ProgID)
```

OLE (.OCX) Custom Controls

OLE custom control Overview

OLE custom controls commonly have the .OCX file extension. Therefore, they are commonly referred to as .OCX controls. .OCX controls are similar to .VBX controls in that they are self-contained and designed to perform a specific task when used in a program. However, .OCX controls do not have the limitations that .VBX controls have, since .OCX controls are built to Microsoft's OLE 2 specification, which was designed with cross-language compatibility in mind (to languages other than just Visual Basic).

.OCX Control Properties

<u>Name</u>	<u>Description</u>
PROP:Create	The CREATE attribute (blank if none). (WRITE ONLY)
PROP:DesignMode	Is the .OCX control in the container in design-mode (does it have a size border around it)? (WRITE ONLY)
PROP:Ctrl	Is this an .OCX control? (READ ONLY)
PROP:GrabHandles	Makes the .OCX control show grab handles. (WRITE ONLY)
PROP:OLE	Is there an OCX or OLE object in the container? (READ ONLY)
PROP:IsRadio	Is this an OCX radio button? (READ ONLY)
PROP>LastEventName	Gets the name of the last event sent to an .OCX control. (READ ONLY)
PROP:SaveAs	Saves the object to an OLE Compound Storage file. (WRITE ONLY) The syntax for placing the object in the file is ' <i>filename\!component</i> ' For example: <code>?controlx{PROP:SaveAs} = 'myfile\!objectx'</code>
PROP:ReportException	Report OLE exceptions (for debug). (WRITE ONLY)
PROP:DoVerb	Executes an OLE doverb command from the following set of commands (WRITE ONLY): DOVERB:Primary (0) Calls the object's primary action. The object, not the container, determines this action. If the object supports in-place activation, the primary verb usually activates the object in-place. DOVERB:Show (-1) Tells the object to show itself for editing or viewing. Called to display newly inserted objects for initial editing and to show link sources. This is usually an alias for some other object-defined action. DOVERB:Open (-2) Tells the object to open itself for editing in a separate window from its container (this includes objects that support in-place activation). If the object does not support in-place activation, this has the same action as DOVERB:Show.

DOVERB:Hide (-3)

Tells the object to remove its user interface. This applies only to objects activated in-place.

DOVERB:UIActivate (-4)

Activates the object in-place, along with its full set of user-interface tools, including menus, toolbars, and its name in the title bar of the container window.

DOVERB:InPlaceActivate (-5)

Activates the object in-place without displaying tools (menus and toolbars) that end-users need to change the behavior or appearance of the object.

DOVERB:DiscardUndoState (-6)

Tells the object to discard any undo state that it may be maintaining without deactivating the object.

DOVERB:Properties (-7)

Invokes the modal system property browser for the object to allow the user to set its properties.

PROP:Language

The number for the language used for OLE Automation or OCX Method. The number for US English is 0409H, and other language numbers can be computed from the data in the WINNT.H file in the MS Windows SDK. (READ/WRITE)

Callback Functions

OCX Event Processor

OCX Property Edit Controller

OCX Property Change

Callback functions are a standard part of Windows programming in most programming languages. A callback function is a PROCEDURE that you (the programmer) write to handle specific situations that the operating system deems the programmer may need to deal with. A callback function is called by the operating system whenever it needs to pass on these situations. Therefore, a callback function does not appear to be part of the logic flow, but instead appears to be separate and "magic" without any logical connection to other procedures in your program.

The Clarion for Windows language does not force you to write your own callback functions for all the common tasks that other programming languages require you to, since the Clarion runtime library and the ACCEPT loop handles most of that for you. However, since .OCX controls are written in other languages that do require callback functions, you will need to write your own to deal with the events and other programming issues for the .OCX controls you use in Clarion programs. Since CLASS methods have an implicit first parameter of the class name, they cannot be used as callbacks.

There are three callback functions you can write for your .OCX controls: an event processor, a property edit controller, and a property change handler. You may name these whatever you want, but they have specific requirements for the parameters that they receive.

OCX Event Processor Callback Function

The prototype for the event processor must be:

```
OcxEventFuncName PROCEDURE( *SHORT, SIGNED, LONG ), LONG
```

The parameters it receives from the operating system are:

- | | |
|--------|---|
| *SHORT | A Reference parameter to pass onto the following other OCX library procedures: OCXGETPARAM, OCXGETPARAMCOUNT, and OCXSETPARAM as their first parameter. |
| SIGNED | The field number for the control. This is the same number that is represented by the control's field equate label. |
| LONG | The number of the .OCX event. Equates for some pre-defined event numbers are contained in the OCXEVENT.CLW file. |

The LONG return value indicates to the operating system whether any further processing is necessary. Returning zero (0) indicates some further processing is necessary (like updating a

USE variable or unchecking a radio button), while returning any other value indicates processing is complete.

Processing the events generated by an .OCX control must occur quickly, since some events have critical timing. Therefore, there should be no user interaction possible within this procedure (such as WINDOWs, ASK statements, or MESSAGE procedures). The code should process only what it needs to, just as quickly as possible (usually, this means eliminating all mouse events).

OCX Property Edit Controller Callback Function

The prototype for the property edit controller must be:

```
OcxPropEditFuncName PROCEDURE(SIGNED,STRING),LONG
```

The parameters it receives from the operating system are:

SIGNED The field number for the control. This is the same as the number represented by the control's field equate label.

STRING The name of the property about to be edited.

The **LONG** return value indicates to the operating system whether permission to edit the property has been granted by the callback function. If the procedure returns zero (0), then permission is denied and the user is not allowed to edit the property. If the procedure returns any value other than zero (0), then permission is granted and the user is allowed to edit the property.

OCX Property Change Callback Function

The prototype for the property change handler must be:

```
OcxPropChangeProcName PROCEDURE(SIGNED,STRING)
```

The parameters it receives from the operating system are:

SIGNED The field number for the control. This is the same as the number represented by the control's field equate label.

STRING The name of the changed property.

This procedure is called when a property has been changed.

Example:

```

! This program uses the Calendar OCX that Microsoft ships with its Access95
! product (specifically, the one in MS Office Professional for Windows 95).
PROGRAM
MAP
INCLUDE('OCX.CLW')
EventFunc PROCEDURE(*SHORT Reference,SIGNED OleControl,LONG CurrentEvent),LONG
PropChange PROCEDURE(SIGNED OleControl,STRING CurrentProp)
PropEdit PROCEDURE(SIGNED OleControl,STRING CurrentProp),LONG
END
INCLUDE('OCXEVENT.CLW') !Constants that OCX events use
INCLUDE('ERRORS.CLW') !Include errorcode constants

GlobalQue QUEUE !Event and change display queue
F1 STRING(255)
END

SaveDate FILE,DRIVER('TopSpeed'),PRE(SAV),CREATE
Record RECORD
DateField STRING(10)
END
END

MainWin WINDOW('OCX Demo'),AT(,,350,200),STATUS(-1,-1),SYSTEM,GRAY,MAX,RESIZE
MENUBAR
MENU('&File')
ITEM('Save Date to File'),USE(?SaveObjectValue)
ITEM('Retrieve Saved Date'),USE(?GetObject)
ITEM('E&xit'),USE(?exit)
END
MENU('&Object')
ITEM('About Box'),USE(?AboutObject)
ITEM('Set Date to TODAY'),USE(?SetObjectValueToday)
ITEM('Set Date to 1st of Month'),USE(?SetObjectValueFirst)
END
ITEM('&Properties!'),USE(?ActiveObj)
END
LIST,AT(237,6,100,100),USE(?List1),HVSCROLL,FROM(GlobalQue)
OLE,AT(5,10,200,150),USE(?OcxObject)
END
END

CODE
OPEN(SaveDate)
IF ERRORCODE() !Check for error on Open
IF ERRORCODE() = NoFileErr !if the file doesn't exist
CREATE(SaveDate) !create it
IF ERRORCODE() THEN HALT(,ERROR()) END

```

```

    OPEN(SaveDate)                                !then open it for use
    IF ERRORCODE() THEN HALT(,ERROR()) END
    ELSE
        HALT(,ERROR())
    END
END
OPEN(MainWin)
?OcXObject{PROP:Create} = 'MSACAL.MSACALCtrl.7' !MS Access 95 Calendar OCX control
IF RECORDS(SaveDate)                                !Check for existing saved record
    SET(SaveDate)                                    !and get it
    NEXT(SaveDate)
    IF ERRORCODE() THEN STOP(ERROR()).
    POST(EVENT:Accepted,?GetObject)
ELSE
    ADD(SaveDate)                                    !or add one
    IF ERRORCODE() THEN STOP(ERROR()).
END
IF ?OcXObject{PROP:OLE}                                !Check for an OLE Object
    GlobalQue = 'An Object is in the OLE control'
    ADD(GlobalQue)
    IF ?OcXObject{PROP:Ctrl}                                !See if Object is an OCX
        GlobalQue = 'It is an OCX Object'
        ADD(GlobalQue)
    END
END
DISPLAY
OCXREGISTEREVENTPROC(?OcXObject,EventFunc)          !Register Event processing Callback
OCXREGISTERPROPCHANGE(?OcXObject,PropChange)        !Register Property Change Callback
OCXREGISTERPROPEEDIT(?OcXObject,PropEdit)           !Register Property Edit Callback
?OcXObject{PROP:ReportException} = 1                !Enable the OCX's error reporting
ACCEPT
CASE EVENT()
OF EVENT:Accepted
    CASE FIELD()
    OF ?Exit
        POST(EVENT:CloseWindow)
    OF ?AboutObject
        ?OcXObject{'AboutBox'}                        !Display control's About Box
    OF ?SetObjectValueToday
        ?OcXObject{'Value'} = FORMAT(TODAY(),@D1)     !Set control to TODAY's date
    OF ?SetObjectValueFirst
        ?OcXObject{'Value'} = MONTH(TODAY()) & '/1/' & SUB(YEAR(TODAY()),3,2)
    OF ?SaveObjectValue
        !Save control's value to file
        SAV:DateField = ?OcXObject{'Value'}
        PUT(SaveDate)
        IF ERRORCODE() THEN STOP(ERROR()).
    OF ?GetObject
        !Set control's value from file
        ?OcXObject{'Value'} = SAV:DateField

```


Calling OLE Object Methods

Both OLE Automation to an OLE Server application and OCX/ActiveX objects publish methods (procedures) that you can call to have the object perform specified actions. Since OCXs are the OLE successors to VBX controls, most OCX vendors provide their example code using Visual Basic (VB) syntax. Those that can be used in C++ programs usually also have C++ code examples.

Translating these examples to the relevant Clarion code usually requires some knowledge of VB or C++. This section demonstrates the most common types of method calls in VB examples and how they translate to Clarion.

Method Syntax Overview

To call any OLE/OCX method, you use Clarion's Property Syntax. You specify the control to which the method or property belongs as the field equate label of the OLE control, then write the method call in a string constant inside the curly braces ({}).

The example code supplied with most OLE controls uses the VB/C++ "dot property" syntax to specify the name of the control and the method to call or the property to set. For example, the following VB code:

```
ControlName.AboutBox
```

translates to Clarion as:

```
?Ole{'AboutBox'}
```

This code displays the "About" dialog for the *ControlName* control. You might also see this example's VB code as:

```
Form1.ControlName.AboutBox
```

This form simply specifies the dialog containing the *ControlName* object. The Clarion translation for this is still the same.

The OLE/OCX object is always referenced in Clarion code by the field equate label of the Clarion OLE control, no matter what the name of the control is in VB, because the object's registered name is specified in the CREATE or OPEN attribute of the OLE control. Therefore, the Clarion runtime library only needs to know the field equate label to know exactly which object is being referenced.

Translating VB's "With" Syntax

Many OLE/OCX code examples use the VB *With ... End With* structure to associate multiple property assignments and/or method calls with a single object. In this case, the object is named in the *With* statement and all the property assignments and method calls within the structure begin with the dot separator, then the name of the property to set or method to call. For example, the following VB code:

```
With Form1.VtChart1
    'displays a 3d chart with 8 columns and 8 rows data
    .chartType = VtChChartType3dBar
    .columnCount = 8
    .rowCount = 8
    For column = 1 To 8
        For row = 1 To 8
            .column = column
            .row = row
            .Data = row * 10
        Next row
    Next column
    'use the chart as the backdrop of the legend
    .ShowLegend = True
End With
```

translates to Clarion as:

```
!displays a 3d chart with 8 columns and 8 rows data
?Ole{'chartType'} = VtChChartType3dBar
?Ole{'columnCount'} = 8
?Ole{'rowCount'} = 8
LOOP column# = 1 TO 8
    LOOP row# = 1 TO 8
        ?Ole{'column'} = column#
        ?Ole{'row'} = row#
        ?Ole{'Data'} = row# * 10
    END
END
!use the chart as the backdrop of the legend
?Ole{'ShowLegend'} = True
```

Since Clarion has no direct equivalent to the VB *With ... End With* structure, you just explicitly name the OLE control's field equate label on each property assignment or method call. The single quote (') in VB code indicates a comment.

VB allows nesting these *With ... End With* structures, so you may need to "travel" back to find the object's name. This example demonstrates nested VB *With* structures:

```
With MyObject
    .Height = 100           ' Same as MyObject.Height = 100.
    .Caption = "Hello World" ' Same as MyObject.Caption = "Hello World".
    With .Font
        .Color = Red       ' Same as MyObject.Font.Color = Red.
        .Bold = True       ' Same as MyObject.Font.Bold = True.
    End With
End With
```

which translates to Clarion as:

```
?Ole{'Height'} = 100           ! MyObject.Height = 100
?Ole{'Caption'} = 'Hello World' ! MyObject.Caption = "Hello World"
?Ole{'Font.Color'} = Red       ! MyObject.Font.Color = Red
?Ole{'Font.Bold'} = True       ! MyObject.Font.Bold = True
```

Parameter Passing to OLE/OCX Methods

Just as in Clarion, there are two ways to pass parameters in VB: by value, or by address (by reference). The VB keywords *ByVal* and *ByRef* specify these two methods in VB code. These terms mean the same thing in VB as in Clarion--passing a parameter by value passes a copy of the contents of the variable, while passing a parameter by reference (VB's default) passes the address of the variable itself so the receiving method can modify its contents.

Using Parentheses

VB syntax can either use parentheses surrounding the parameter list or not. If the VB method does not return a value, or you don't care about the returned value, the parameters are passed in VB without parentheses, like this:

```
VtChart1.InsertColumns 6,3
```

If you do want the returned value, then the parameters are passed in VB within parentheses, like this:

```
ReturnValue = VtChart1.InsertColumns (6,3)
```

In Clarion syntax, parameters are always passed within parentheses. Therefore, these two examples translate to:

```
?Ole{'InsertColumns(6,3)'}
ReturnValue = ?Ole{'InsertColumns(6,3)'}
```

Passing Parameters By Value

Value parameters are passed to OLE/OCX objects as strings (except Boolean parameters). Since OLE/OCX objects are supposed to cast their input to the correct data types using a VARIANT mechanism (similar to Clarion's data type conversion), this allows the most compatibility with the least work. Any string which requires a double quote mark (") needs to include two (").

Value parameters may be passed to OLE/OCX object methods as constants or variables. The examples above pass parameters as constants. You may not have blank spaces in the constant unless the parameter is contained in double quotes (for example, "Value with blanks").

There are two ways to pass a Clarion variable to an OLE/OCX method by value: concatenated into the string constant that calls the method, or by using BIND on the variable name and placing the name of the variable directly in the string constant that calls the method. For example, to rewrite the above example passing the variable values in a concatenated string:

```
ColumnNumber = 6
NumberOfColumns = 3
?Ole{'InsertColumns(' & ColumnNumber & ',' & NumberOfColumns & ')'}
!Same as ?Ole{'InsertColumns(6,3)'}
```

The second way to pass variables by value is to BIND them and name them in the string constant, like this:

```

BIND('ColumnNumber',ColumnNumber)
BIND('NumberOfColumns',NumberOfColumns)
?Ole{'InsertColumns(ColumnNumber,NumberOfColumns)'}
!Same as ?Ole{'InsertColumns(6,3)'}

```

This method makes the code more easily readable, but you must first BIND the variables to pass.

Passing Parameters By Address (Reference)

Parameters passed by address may be passed to OLE/OCX object methods only as named variables in the constant string. Therefore, you must use BIND on the variable name and place the name of the variable directly in the string constant that calls the method with an ampersand prepended to the variable name to signal that the variable is being passed by reference. For example, to re-write the above example to pass the variables by address:

```

ColumnNumber = 6
NumberOfColumns = 3
BIND('ColumnNumber',ColumnNumber)
BIND('NumberOfColumns',NumberOfColumns)
?Ole{'InsertColumns(&ColumnNumber,&NumberOfColumns)'}

```

Parameters passed by address are passed to OLE/OCX objects as the data type of the bound variable (except Boolean parameters). The variables are actually passed as temporary string variables which the Clarion library automatically dereferences so that any modifications to the passed variable by the OLE/OCX method are carried back to the original variable passed.

Boolean Parameters

Boolean parameters (1/0 or True/False) are passed either by value or by address. When passing by value, you may either pass a constant (a 1 or 0, or the words TRUE or FALSE), like this:

```

?Ole{'ODBCConnect(&DataSource,1,&RetVal)'}
?Ole{'ODBCConnect(&DataSource,TRUE,&RetVal)'}

```

or pass a variable name (after BINDing it) within a "bool()" call, like this:

```

BoolParm = 1
BIND('BoolParm',BoolParm)
?Ole{'ODBCConnect(&DataSource,bool(BoolParm),&RetVal)'}

```

Bool() is a construct that tells the property expression parser to pass it as a Boolean value. Bool() is only valid within an OLE/OCX method call string.

To pass by reference, simply prepend an ampersand to the variable name within the bool() construct, like this:

```
    BIND( 'BoolParm', BoolParm)
    ?Ole{ 'ODBCConnect(&DataSource, bool(&BoolParm), &RetVal) ' }
```

Named Parameters

In VB, there are two ways to pass parameters: positionally, or as "named arguments." Positional parameters imply that you must either pass a parameter or place a comma place-holder for any omitted parameters in the method call. Since some methods can receive a large number of parameters, this can result in a long string of comma place-holders when you simply want to pass one or two parameters to the method. VB solves this problem by allowing programmers to "name" the parameters, which allows the programmer calling the method to only pass the few parameters they choose to without regard to their position or order within the parameter list.

Named parameters are not universally supported in VB, so the OLE/OCX vendor needs to have written their methods specifically to support them. The OLE/OCX help file should state whether named parameters are supported, or you can use VB's Object Browser to determine whether they are supported and the parameter names to use.

The VB syntax for named parameters uses := to assign the value to the parameter's name. For example, for the following VB statement:

```
    OpenIt(Name:=, [Exclusive]:=, [ReadOnly]:=, [Connect]:=)
```

you can call the method in VB using positional parameters, like this:

```
    Db = OpenIt("MyFile", False, False, "ODBC;UID=Fred")
```

which translates to Clarion (using positional parameters) as:

```
    Db = ?Ole{ 'OpenIt("MyFile", False, False, "ODBC;UID=Fred") ' }
```

You can call the same method in VB using named parameters, like this (the underscore character is VB's line continuation character):

```
    Db = OpenIt(Name:="MyFile", Exclusive:=False, ReadOnly:=False, _ Connect:="ODBC;U
```

which translates to Clarion as:

```
    Db = ?Ole{ 'OpenIt(Name="MyFile", Exclusive=False, ReadOnly=False, ' & |
    'Connect="ODBC;UID=Fred") ' }
```

or you can pass the parameters in VB in a different order:

```
    Db = OpenIt(Connect:="ODBC;UID=Fred", _
    Name:="MyFile", _
    ReadOnly:=False, _
    Exclusive:=False)
```

which translates to Clarion as:

```
    Db = ?Ole{ 'OpenIt(Connect="ODBC;UID=Fred", Name="MyFile", ' & |
    'Exclusive=False, ReadOnly=False') ' }
```

OCX Library Procedures

OCXREGISTERPROPEDIT (install property edit callback)

OCXREGISTERPROPEDIT(*control* , *procedure*)

OCXREGISTERPROPEDIT

Installs a property edit callback function.

control An integer expression containing the field number or field equate label of the OLE control to affect.

procedure The label of the property edit callback function for the *control*.

OCXREGISTERPROPEDIT installs a property edit callback *procedure* for the *control*. The callback function *procedure* controls property edits to the *control* by allowing or disallowing them.

Example:

```
OCXREGISTERPROPEDIT( ?OleControl , CallbackFunc )
```

See Also:

Callback Functions

OCXREGISTERPROPCHANGE (install property change callback)

OCXREGISTERPROPCHANGE(*control* , *procedure*)

OCXREGISTERPROPCHANGE

Installs a property change callback procedure.

control An integer expression containing the field number or field equate label of the OLE control to affect.

procedure The label of the property change callback procedure for the *control*.

OCXREGISTERPROPCHANGE installs a property change callback *procedure* for the *control*. The callback *procedure* is called when a property of the *control* has been changed.

Example:

```
OCXREGISTERPROPCHANGE(?OleControl,CallbackProc)
```

See Also:

Callback Functions

OCXREGISTEREVENTPROC (install event processing callback)

OCXREGISTEREVENTPROC(*control* , *procedure*)

OCXREGISTEREVENTPROC

Installs an OCX event callback procedure.

control An integer expression containing the field number or field equate label of the OLE control to affect.

procedure The label of the event processing callback procedure for the *control*.

OCXREGISTEREVENTPROC installs an OCX event callback *procedure* for the *control*. The callback *procedure* is called when any event is posted by the operating system for the *control*.

Example:

```
OCXREGISTEREVENTPROC(?OleControl,CallbackProc)
```

See Also:

Callback Functions

OCXUNREGISTERPROPEDIT (un-install property edit callback)

OCXUNREGISTERPROPEDIT(*control*)

OCXUNREGISTERPROPEDIT

Un-installs a property edit callback procedure.

control An integer expression containing the field number or field equate label of the OLE control to affect.

OCXUNREGISTERPROPEDIT un-installs a property edit callback *procedure* for the *control*.

Example:

```
OCXUNREGISTERPROPEDIT(?OleControl)
```

See Also:

Callback Functions

OCXUNREGISTERPROPCHANGE (un-install prop change callback)

OCXUNREGISTERPROPCHANGE(*control*)

OCXUNREGISTERPROPCHANGE

Un-installs a property change callback procedure.

control An integer expression containing the field number or field equate label of the OLE control to affect.

OCXUNREGISTERPROPCHANGE un-installs a property change callback *procedure* for the *control*.

Example:

```
OCXUNREGISTERPROPCHANGE( ?OleControl )
```

See Also:

Callback Functions

OCXUNREGISTEREVENTPROC (un-install event process callback)

OCXUNREGISTEREVENTPROC(*control*)

OCXUNREGISTEREVENTPROC

Un-installs an OCX event callback procedure.

control An integer expression containing the field number or field equate label of the OLE control to affect.

OCXUNREGISTEREVENTPROC un-installs an OCX event callback *procedure* for the *control*.

Example:

```
OCXUNREGISTEREVENTPROC(?OleControl)
```

See Also:

Callback Functions

OCXGETPARAMCOUNT (return number of parameters for current event)**OCXGETPARAMCOUNT**(*reference*)**OCXGETPARAMCOUNT**

Returns the number of parameters associated with the current OCX event.

reference The label of the first parameter of the event processing callback procedure.**OCXGETPARAMCOUNT** returns the number of parameters associated with the current .OCX event. This procedure is only valid when the .OCX event processing callback function is active.

Return Data Type: USHORT

Example:

```

OEvent  PROCEDURE(Reference,OleControl,CurrentEvent)  !Event processing callback proc
Count   LONG
Res     CSTRING(200)
Parm    CSTRING(30)
CODE
IF CurrentEvent <> OCXEVENT:MouseMove                !Eliminate mouse move events
  Res = 'Control ' & OleControl & ' Event ' & OleControl{PROP>LastEventName} & ':'
  LOOP Count = 1 TO OCXGETPARAMCOUNT(Reference)      !Cycle through all parameters
    Parm = OCXGETPARAM(Reference,Count)                !getting each parameter name
    Res = CLIP(Res) & ' ' & Parm                       !and concatenate them together
  END
  GlobalQue = Res                                     !Assign to a global QUEUE
  ADD(GlobalQue)                                     !add the entry for later display
END
RETURN(True)                                         !of all the OCX events and their
                                                    !parameters

```

See Also:

Callback Functions

OCXGETPARAM

OCXGETPARAM (return current event parameter string)**OCXGETPARAM**(*reference* , *number*)**OCXGETPARAM** Returns the value of a parameter associated with the current OCX event.*reference* The label of the first parameter of the event processing callback procedure.*number* The number of the parameter to retrieve.**OCXGETPARAM** returns the value of the *number* parameter associated with the current .OCX event. This procedure is only valid when the .OCX event processing callback function is active.Return Data Type: **STRING**

Example:

```

OEvent PROCEDURE(Reference,OleControl,CurrentEvent) !Event processing callback proc
Count LONG
Res CSTRING(200)
Parm CSTRING(30)
CODE
IF CurrentEvent <> OCXEVENT:MouseMove !Eliminate mouse move events
Res = 'Control ' & OleControl & ' Event ' & OleControl{PROP:LastEventName} & ':'
LOOP Count = 1 TO OCXGETPARAMCOUNT(Reference) !Cycle through all parameters
Parm = OCXGETPARAM(Reference,Count) !getting each parameter name
Res = CLIP(Res) & ' ' & Parm !and concatenate them together
END
GlobalQue = Res !Assign to a global QUEUE
ADD(GlobalQue) !add the entry for later display
END !of all the OCX events and their
RETURN(True) !parameters

```

See Also:

Callback Functions

OCXSETPARAM

OCXGETPARAMCOUNT

OCXSETPARAM (set current event parameter string)

OCXSETPARAM(*reference* ,*number* ,*value*)

OCXSETPARAM	Sets the value of a parameter associated with the current OCX event.
<i>reference</i>	The label of the first parameter of the event processing callback procedure.
<i>number</i>	The number of the parameter to set.
<i>value</i>	A string constant or variable containing the value to set.

OCXSETPARAM sets the value of the *number* parameter associated with the current event. This is only allowed on parameters that are passed by address (see the OCX control's documentation for valid parameters to set). If the modification is not allowed it will be ignored. This procedure is only valid when the .OCX event processing callback function is active.

Example:

```

OEvent  PROCEDURE(Reference,OleControl,CurrentEvent)  !Event processing callback proc
Count  LONG
Res    CSTRING(200)
Parm   CSTRING(30)
CODE
IF CurrentEvent <> OCXEVENT:MouseMove                !Eliminate mouse move events
  Res = 'Control ' & OleControl & ' Event ' & OleControl{PROP:LastEventName} & ':'
  LOOP Count = 1 TO OCXGETPARAMCOUNT(Reference)    !Cycle through all parameters
    Parm = OCXGETPARAM(Reference,Count)              !getting each parameter name
    Res = CLIP(Res) & ' ' & Parm                     !and concatenate them together
    OCXSETPARAM(Reference,1,'1')                    !Change the parameter's value
  END
  GlobalQue = Res                                    !Assign to a global QUEUE
  ADD(GlobalQue)                                     !add the entry for later display
END                                                  !of all the OCX events and their
RETURN(True)                                        !parameters

```

See Also:

Callback Functions

OCXGETPARAM

OCXLOADIMAGE (return an image object)**OCXLOADIMAGE**(*name*)

OCXLOADIMAGE

Returns an image object.

name A string expression containing the name of the file or resource to load.**OCXLOADIMAGE** returns an image object. This image object can be assigned to any control that uses an image object (such as a VB imagelist control).Return Data Type: **STRING**

Example:

```
!Add an image to an ImageList control:  
?imagelist{ 'ListImages.Add(,,' & OCXLOADIMAGE('CLOCK.BMP') & ')'
```

Appendix B - Events

Events

In Clarion Windows programs, most of the messages from Windows are automatically handled internally by the ACCEPT event processor. These are the common events handled by the runtime library (screen re-draws, etc.). Only those events that actually may require program action are passed on by ACCEPT to your Clarion code. The net effect of this is to make your programming job easier by removing the low-level "drudgery" code from your program, allowing you to concentrate on the high-level aspects of programming, instead. Of course, it is also possible to handle these low-level messages yourself by "sub-classing" the window, but that is a low-level technique that should only be used if absolutely necessary. Consult Charles Petzold's book *Programming Windows* published by Microsoft Press if you need more information on sub-classing.

There are two types of events passed on to the program by ACCEPT: **Field-specific** and **Field-independent** events. The following lists are the event EQUATES that are contained in EQUATES.CLW.

Field-Independent Events

Field-Specific Events

Field-Independent Events

A **Field-independent** event does not relate to any one control but requires some program action (for example, to close a window, quit the program, or change execution threads). Most of these events cause the system to become modal for the period during which they are processing, since they require a response before the program may continue.

EVENT:AlertKey	EVENT:BuildDone
EVENT:BuildFile	EVENT:BuildKey
EVENT:CloseDown	EVENT:CloseWindow
EVENT:Completed	DDE Events
EVENT:Docked	EVENT:Undocked
EVENT:GainFocus	EVENT:Iconize
EVENT:Iconized	EVENT:LoseFocus
EVENT:Maximize	EVENT:Maximized
EVENT:Move	EVENT:Moved
EVENT:OpenWindow	EVENT:PreAlertKey
EVENT:Restore	EVENT:Restored
EVENT:Resume	EVENT:Size
EVENT:Sized	EVENT:Suspend
EVENT:Timer	

EVENT:AlertKey

The user pressed an ALRT attribute (or ALERT statement) hot key for an ALRT attribute on the window. This is the event on which you perform the action the user has requested by pressing the alert key.

EVENT:BuildDone

The BUILD or PACK statement has completed re-building the keys. This is the event on which you perform any build cleanup code. If the user cancelled the BUILD, ERRORCODE 93 is set.

EVENT:BuildFile

The BUILD or PACK statement is re-building the file. This is the event on which you inform your user of the progress of the build.

EVENT:BuildKey

The BUILD or PACK statement is re-building the key. This is the event on which you inform your user of the progress of the build.

EVENT:CloseDown

The application is closing. POSTing this event closes the application. This is the event on which you perform any application cleanup code.

EVENT:CloseWindow

The window is closing. POSTing this event closes the window. This is the event on which you perform any window cleanup code.

EVENT:Completed

AcceptAll (non-stop) mode has finished processing all the window's controls. This is the event on which you have executed all data entry validation code for the controls in the window and can safely write to disk.

DDE Events

EVENT:DDEadvise

A client has requested continuous updates of a data item from this Clarion DDE server application. This is the event on which you execute DDEWRITE to provide the data to the client every time it changes.

EVENT:DDEclosed

A DDE server has terminated the DDE link to this Clarion client application.

EVENT:DDEdata

A DDE server has supplied an updated data item to this Clarion client application.

EVENT:DDEexecute

A client has sent a command to this Clarion DDE server application (if the client is another Clarion application, it has executed a DDEEXECUTE statement). This is the event on which you determine the action the client has requested and perform it, then execute a CYCLE statement to signal positive acknowledgement to the client that sent the command.

EVENT:DDEpoke

A client has sent unsolicited data to this Clarion DDE server application. This is the event on which you determine what the client has sent and where to place it, then execute a CYCLE statement to signal positive acknowledgement to the client that sent the data.

EVENT:DDErequest

A client has requested a data item from this Clarion DDE server application. This is the event on which you execute DDEWRITE to provide the data to the client once.

EVENT:Docked

A dockable toolbox window has been docked or its docked position has been changed.

EVENT:Undocked

A dockable toolbox window has been undocked.

EVENT:GainFocus

The window is gaining input focus from another thread. This is the event on which you restore any data you saved in EVENT:LoseFocus. The system is modal during this event. EVENT:GainFocus is not generated until EVENT:LoseFocus is processed (if focus was on another window of the same program).

EVENT:Iconize

The user is minimizing a window with the IMM attribute. If a CYCLE statement is encountered in the code to process this event, the EVENT:Iconized is not generated and the action is aborted. This is the event on which you can prevent users from minimizing a window. The system is modal during this event.

EVENT:Iconized

The user has minimized a window with the IMM attribute. This is the event on which you readjust anything that is screen-size-dependent.

EVENT:LoseFocus

The window is losing input focus to another thread. This is the event on which you save any data that could be at risk of being changed by another thread. The system is modal during this event.

EVENT:Maximize

The user is maximizing a window with the IMM attribute. If a CYCLE statement is encountered in the code to process this event, EVENT:Maximized is not generated and the action is aborted. This is the event on which you can prevent users from maximizing a window. The system is modal during this event.

EVENT:Maximized

The user has maximized a window with the IMM attribute. This is the event on which you readjust anything that is screen-size-dependent.

EVENT:Move

The user is moving a window with the IMM attribute. If a CYCLE statement is encountered in the code to process this event, EVENT:Moved is not generated and the action is aborted. This is the event on which you can prevent users from moving a window. The system is modal during this event.

EVENT:Moved

The user has moved a window with the IMM attribute. This is the event on which you readjust anything that is screen-position-dependent.

EVENT:Notify

A NOTIFY statement has been executed. This event is used by the receiver thread to detect that a NOTIFY has been sent.

EVENT:OpenWindow

The window is opening. This is the event on which you perform any window initialization code.

EVENT:PreAlertKey

The user pressed an ALRT attribute (or ALERT statement) hot key for an ALRT attribute on the window. If a CYCLE statement executes in the code to process this event, the normal library action for the keystroke executes before EVENT:AlertKey generates. This event allows you to specify whether the normal library action for the keystroke executes or not, in addition to the code you place in EVENT:AlertKey. The system is modal during this event.

EVENT:Restore

The user is restoring the previous size of a window with the IMM attribute. If a CYCLE statement is encountered in the code to process this event, EVENT:Restored is not generated and the action is aborted. This is the event on which you can prevent users from restoring a window. The system is modal during this event.

EVENT:Restored

The user has restored the previous size of a window with the IMM attribute. This is the event on which you readjust anything that is screen-size-dependent.

EVENT:Resume

The window still has input focus and is regaining control from an EVENT:Suspend. The system is modal during this event.

EVENT:Size

The user is resizing a window with the IMM attribute. If a CYCLE statement is encountered in the code to process this event, EVENT:Sized is not generated and the action is aborted. This is the event on which you can prevent users from resizing a window. The system is modal during this event.

EVENT:Sized

The user has resized a window with the IMM attribute. This is the event on which you readjust anything that is screen-size-dependent.

EVENT:Suspend

The window still has input focus but is giving control to another thread to process timer events. The system is modal during this event.

EVENT:Timer

The TIMER attribute has triggered. This is the event on which you perform any timed actions, such as clock display, or background record processing for reports or batch processes.

Field-Specific Events

Field-Specific Events

A **Field-specific** event occurs when the user presses a key that may require the program to perform a specific action related to that control.

EVENT:Accepted	EVENT:AlertKey	EVENT:ColumnResize
EVENT:Contracted	EVENT:Contracting	EVENT:Drag
EVENT:Dragging	EVENT:Drop	EVENT:DroppedDown
EVENT:DroppingDown	EVENT:Expanded	EVENT:Expanding
EVENT:Locate	(Mouse Events)	EVENT:NewSelection
EVENT:PageDown	EVENT:PageUp	EVENT:PreAlertKey
EVENT:Rejected	EVENT:ScrollBottom	EVENT:ScrollDown
EVENT:ScrollDrag	EVENT:ScrollTop	EVENT:ScrollTrack
EVENT:ScrollUp	EVENT:Selected	EVENT:TabChanging

EVENT:Accepted

The user has entered data or made a selection then pressed TAB or CLICKED the mouse to move on to another control. This is the event on which you should perform any data input validation code.

EVENT:AlertKey

The user pressed an ALRT attribute hot key for an ALRT attribute on the control. This is the event on which you perform the action the user has requested by pressing the alert key.

EVENT:ColumnResize

On a LIST control with an M in the FORMAT attribute string, the user has resized a column.

EVENT:Contracted

On a LIST control with T in the FORMAT attribute string, the user has clicked on a tree expansion box.

EVENT:Contracting

On a LIST control with T in the FORMAT attribute string, the user has clicked on a tree contraction box. If a CYCLE statement is encountered in the code to process this event, the EVENT:Contracted is not generated and the contraction is aborted. The system is modal during this event.

EVENT:Drag

The user released the mouse button over a valid drop target. This event is posted to the control from which the user is dragging. This is the event on which you set the program to pass the dragged data to the drop target.

EVENT:Dragging

The user is dragging the mouse from a control with the DRAGID attribute and the mouse cursor is over a valid potential drop target. This event is posted to the control from which the user is dragging. This is the event on which you can change the mouse cursor to indicate a valid drop target.

EVENT:Drop

The user released the mouse button over a valid drop target. This event is posted to the drop target control. This is the event on which you receive the dragged data.

EVENT:DroppedDown

On a LIST or COMBO control with the DROP attribute, the list has dropped. This is the event on which you can hide other controls that the droplist covers to prevent "screen clutter" from distracting the user.

EVENT:DroppingDown

On a LIST or COMBO control with the DROP attribute, the user pressed the down arrow button. This is the event on which you get the records when "demand-loading" the list.

EVENT:Expanded

On a LIST control with T in the FORMAT attribute string, the user has clicked on a tree expansion box.

EVENT:Expanding

On a LIST control with T in the FORMAT attribute string, the user has clicked on a tree expansion box. If a CYCLE statement is encountered in the code to process this event, the EVENT:Expanded is not generated and the expansion is aborted. The system is modal during this event.

EVENT:Locate

On a LIST control with the VCR attribute, the user pressed the locator (?) VCR button. This is the event on which you can unhide the locator entry control, if it is kept hidden.

Mouse Events

EVENT:MouseDown

On a REGION with the IMM attribute, a synonym for EVENT:Accepted (for code readability, only).

EVENT:MouseIn

On a REGION with the IMM attribute, the mouse cursor has entered the region.

EVENT:MouseMove

On a REGION with the IMM attribute, the mouse cursor has moved within the region.

EVENT:MouseOut

On a REGION with the IMM attribute, the mouse cursor has left the region.

EVENT:MouseUp

On a REGION with the IMM attribute, the mouse button has been released.

EVENT:NewSelection

In a LIST, COMBO, SHEET, or SPIN control, this event generates when the current selection has changed. In an ENTRY control with the IMM attribute, this event generates whenever the contents of the control changes or the cursor moves. This is the event on which you perform any "housekeeping" to synchronize other controls with the currently highlighted record in the list, or determine that the user has entered all allowable data in the ENTRY.

EVENT:PageDown

On a LIST or COMBO control with the IMM attribute, the user pressed PGDN. This is the event on which you get the next page of records when "page-loading" the list.

EVENT:PageUp

On a LIST or COMBO control with the IMM attribute, the user pressed PGUP. This is the event on which you get the previous page of records when "page-loading" the list.

EVENT:PreAlertKey (Field Specific)

The user pressed an ALRT attribute hot key for an ALRT attribute on the control. If a CYCLE statement is encountered in the code to process this event, the normal library action for the keystroke executes before EVENT:AlertKey generates. This event allows you to specify whether the normal library action for the keystroke executes or not, in addition to the code you place in EVENT:AlertKey. The system is modal during this event.

EVENT:Rejected

The user has entered an invalid value for the entry picture, or an out-of-range number on a SPIN control. The REJECTCODE procedure returns the reason the user's input has been rejected and you can use the PROP:ScreenText property to get the user's input from the screen. This is the event on which you alert the user to the exact problem with their input.

EVENT:ScrollBottom

On a LIST or COMBO control with the IMM attribute, the user pressed CTRL+PGDN. This is the event on which you get the last page of records when "page-loading" the list.

EVENT:ScrollDown

On a LIST or COMBO control with the IMM attribute, the user has attempted to move the highlight bar down in the LIST. This is the event on which you get the next record when "page-loading" the list or just move the highlight bar when getting another record isn't needed.

EVENT:ScrollDrag

On a LIST or COMBO control with the IMM attribute, the user has moved the scroll bar's "thumb" and has just released the mouse button. This is the event on which you dynamically scroll the displayed records based on the current value of PROP:VScrollPos.

EVENT:ScrollTop

On a LIST or COMBO control with the IMM attribute, the user pressed CTRL+PGUP. This is the event on which you get the first page of records when "page-loading" the list.

EVENT:ScrollTrack

On a LIST or COMBO control with the IMM attribute, the user is currently moving the scroll bar's "thumb." This is an event on which you can dynamically scroll the displayed records based on the current value of PROP:VScrollPos.

EVENT:ScrollUp

On a LIST or COMBO control with the IMM attribute, the user has attempted to move the highlight bar up in the LIST. This is the event on which you get the previous record when "page-loading" the list or just move the highlight bar when getting another record isn't needed.

EVENT:Selected

The control has received input focus. This is the event on which you should perform any data initialization code.

EVENT:TabChanging

On a SHEET control, focus is about to pass to another tab. This is the event on which you perform any necessary "housekeeping" code for the tab you're leaving.

Modal Events

The following Events must always be treated as "modal":

EVENT:LoseFocus	EVENT:GainFocus	EVENT:Dragging
EVENT:Contracting	EVENT:Expanding	EVENT:PreAlertKey
EVENT:TabChanging	EVENT:Move	EVENT:Size
EVENT:Restore	EVENT:Maximize	EVENT:Iconize
EVENT:Sized	EVENT:Selected	

Also, if a list box has the IMM attribute

```
EVENT:ScrollUp
EVENT:ScrollDown
EVENT:PageUp
EVENT:PageDown
EVENT:ScrollTop
EVENT:ScrollBottom
EVENT:ScrollTrack
EVENT:ScrollDrag
```

When we refer to an event as "modal", it means that until the event is completed the runtime library can't continue its normal work. The main rule for writing code that responds to these events is that the event must be completed within a "finite" period of time.

This means that the code you write for handling "modal" events should not invoke dialogs, include inner ACCEPT loops, call Message Boxes, or anything similar that would prevent the ACCEPT loop from completing the processing of the event. When your program needs to do something like this upon processing one of these events, a custom event should be posted to the front of the event queue. For example:

```
CASE EVENT( )
...
OF EVENT:GainFocus
    POST(MyGainFocus,,, TRUE) !Post a user defined event to handle GainFocus action
...
OF MyGainFocus
    <perform actions for gaining focus>
...
END
```

Appendix C - Runtime Properties

PROP:AcceptAll

Property of a WINDOW which returns one (1) if AcceptAll mode is active and zero (0) if it is not, and may also be used to toggle AcceptAll (non-stop) mode. SELECT with no parameters usually initiates AcceptAll mode. This is a field edit mode in which each control in the window is processed by generating EVENT:Accepted for each. This allows data entry validation code to execute for all controls, including those that the user has not touched. AcceptAll mode immediately terminates when any of the following conditions is met:

```
SELECT(?)
Window{PROP:AcceptAll} = 0
A REQ control is left blank or zero.
```

The SELECT(?) statement selects the same control for the user to edit. This code usually indicates the value it contains is invalid and the user must re-enter data. The Window{PROP:AcceptAll} = 0 statement toggles AcceptAll mode off. Assigning values to this property can be used to initiate and terminate AcceptAll mode. When a control with the REQ attribute is left blank or zero, AcceptAll mode terminates with the control highlighted for user entry, without processing any more fields in the TAB key sequence. EVENT:Completed is posted to the window when all controls have been successfully processed.

Example:

```
Screen WINDOW,PRE(Scr)
  ENTRY(@N3),USE(Ctl:Code)
  ENTRY(@S30),USE(Ctl:Name),REQ
  BUTTON('OK'),USE(?OkButton),KEY(EnterKey)
  BUTTON('Cancel'),USE(?CanxButton),KEY(EscKey)
END
CODE
OPEN(Screen)
ACCEPT
  IF EVENT() = EVENT:Completed THEN BREAK.  !AcceptAll mode terminated
  CASE ACCEPTED()
  OF ?Ctl:Code
    IF Ctl:Code > 150                          !If data entered is invalid
      BEEP                                     ! alert the user and
      SELECT(?)                               ! make them re-enter the data
    END
  OF ?OkButton
    Screen{PROP:AcceptAll} = 1                !Initiate AcceptAll mode
  END
END
!Terminate ACCEPT and CASE ACCEPTED
```

PROP:Active

WINDOW property which returns 1 if the window is the active window, blank if not. Set to 1 to make the top window of a thread the active window.

Example:

```

CODE
OPEN(ThisWindow)
X# = START(AnotherThread)      !Start another thread
ACCEPT
CASE EVENT()
OF EVENT:LoseFocus             !When this window is losing focus
  IF Y# <> X#                   ! check for the first focus change
    ThisWindow{PROP:Active} = 1 ! and return focus to this thread
    Y# = X#                     ! then flag first focus change completed
  END
END
END
END

```

PROP:AlwaysDrop

When set to zero, the drop portion of a COMBO or LIST control with the DROP attribute only drops down when the user clicks on the dropdown icon, and when the user presses the down arrow key the displayed entries rotate without dropping down the list. When set to anything other than zero, the drop portion drops down either on the down arrow key or click on the dropdown icon.

Example:

```

MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
  COMBO(@S20),AT(0,0,20,220),USE(MyCombo),FROM(Que),DROP(5)
  END
CODE
OPEN(MDIChild)
?MyCombo{PROP:AlwaysDrop} = 0 !Set windows-like drop behavior

```

PROP:AppInstance

Returns the instance handle (HInstance) of the .EXE file for use in low-level API calls which require it. This is only used with the SYSTEM built-in variable. (READ-ONLY)

Example:

```
PROGRAM
HInstance LONG
CODE
OPEN(AppFrame)
HInstance = SYSTEM{PROP:AppInstance} !Get .EXE instance handle for later use
ACCEPT
END
```

PROP:AssertHook

A property of the SYSTEM built-in variable that sets the override procedure for the ASSERT internal Clarion procedure. Equivalent to {PROP:LibHook,8}. Assign the ADDRESS of the overriding procedure, and the runtime library will call the overriding procedure instead of the ASSERT procedure. Assign zero and the runtime library will once again call its internal procedure. The overriding procedure's prototype must be exactly the same as the ASSERT procedure (STRING filename,UNSIGNED LineNumber). (WRITE-ONLY)

See PROP:AssertHook2 for example source.

PROP:AutoPaper

Sets and returns the state of the automatic best-fit paper selection feature of the report engine. The default value is feature-enabled. This is only used with the SYSTEM built-in variable.

Example:

```
PROGRAM
CODE
OPEN(AppFrame)
SYSTEM{PROP:AutoPaper} = '' !Turn off best-fit paper selection
ACCEPT
END
```

PROP:BreakVar

Sets the variable for a BREAK structure within a REPORT.

Example:

```
Report
REPORT,AT(1000,2000,6000,7000),PRE(RPT),FONT('Arial',10,,FONT:regular),THOUS
Break1  BREAK(ORD:CustNumber),USE(?Break1)
Break2  BREAK(DTL:OrderNumber),USE(?Break2)
        HEADER,AT(,,,2167)
        END
detail  DETAIL,AT(,,,385)
        END
        FOOTER,PAGEAFTER(-1)
        END
        END
        FOOTER,PAGEAFTER(-1),AT(,,,1385)
        END
        END
        FOOTER,AT(1000,9000,6000,1000)
        END
END

CODE
OPEN(Report)
Report$?Break1{PROP:BreakVar} = ORD:CustName !Change the break variable for
Break1
```

PROP:Buffer

Property of a window which allows you to select off-display background re-paints. This can dramatically reduce screen flicker in some situation (such as animated GIF images), but incurs a potentially large memory overhead.

The default value is zero (0) which draws directly to the screen. This is fastest and incurs no memory overhead, but may create flicker in some cases.

Assigning one (1) allocates a permanent memory buffer for the window. This is quite fast, but incurs the most memory overhead.

Assigning two (2) reallocates a memory buffer for the window each time a re-paint is required. This is slower, but incurs the least memory overhead while still reducing flicker.

Example:

```
WinView WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
  END
  CODE
  OPEN(WinView)
  WinView{PROP:Buffer} = 1    !Permanent redraw buffer for the window
```

PROP:Checked

Returns the current display state of a CHECK control--checked (1) or un-checked (""). (READ ONLY)

Example:

```
WinView WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
  CHECK('Check Me'),AT(20,0,20,20),USE(CheckVar)
  END

  CODE
  OPEN(WinView)
  IF ?CheckVar{PROP:Checked} = TRUE  !Is it checked?
    !Do something
  END
  ACCEPT
  END
```

PROP:Child and PROP:ChildIndex

PROP:Child is an array property which returns the control number of a child control in a parent control structure (such as an TAB, OPTION, or GROUP). (READ ONLY) The element number is the ordinal position of the control in the parent structure. Returns null string (") if the element number is out of range.

PROP:ChildIndex is an array property which returns the ordinal position of all the child controls in a parent control structure (such as an TAB, OPTION, or GROUP). (READ ONLY) The element number is the control number of the control in the parent structure. Returns a null string (") if the element number is out of range.

Example:

```
WinView WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
    OPTION('Option 1'),USE(OptVar1),KEY(F10Key),HLP('Option1Help')
        RADIO('Radio 1'),AT(0,0,20,20),USE(?R1)
        RADIO('Radio 2'),AT(20,0,20,20),USE(?R2)
    END
END
CODE
OPEN(WinView)
LOOP X# = 1 TO 99
    Y# = ?OptVar1{PROP:Child,X#}      !Get field numbers of controls in OPTION
    IF NOT Y# THEN BREAK.
    Z# = ?OptVar1{PROP:ChildIndex,Y#} !Get ordinal position of controls in
OPTION
    MESSAGE('Radio ' & Z# & ' is field number ' & Y#)
END
ACCEPT
END
```

PROP:ChoiceFeg

Returns or sets the field number of the currently selected TAB in a SHEET, or RADIO in an OPTION structure.

Example:

```
WinView WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
    OPTION('Option 1'),USE(OptVar1),KEY(F10Key),HLP('Option1Help')
    RADIO('Radio 1'),AT(0,0,20,20),USE(?R1)
    RADIO('Radio 2'),AT(20,0,20,20),USE(?R2)
    END
END

CODE
OPEN(WinView)
?OptVar1{PROP:ChoiceFeg} = ?R1 !Select radio one
ACCEPT
END
```

PROP:ClientHandle

WINDOW property which returns the client window handle (the area of the window that contains the controls) for use with low-level Windows API calls that require it. (READ-ONLY)

Example:

```
WinView WINDOW('View'),AT(0,0,320,200),MDI,MAX,HVSCROLL
    END
MessageText CSTRING('You cannot exit the program from this window ')
MessageCaption CSTRING('No EVENT:CloseDown Allowed ')
TextAddr LONG
CaptionAddr LONG
RetVal SHORT
CODE
OPEN(WinView)
ACCEPT
CASE EVENT()
OF EVENT:CloseDown
    TextAddr = ADDRESS(MessageText)
    CaptionAddr = ADDRESS(MessageCaption)
    RetVal = MessageBox(WinView{PROP:ClientHandle},TextAddr,CaptionAddr,MB_OK)
    !Windows API call using a window handle
CYCLE
    !Disallow program closedown from this window
END
END
```

PROP:ClientWndProc

WINDOW property which sets or gets the client window (not including title or status areas) messaging procedure for use with low-level Windows API calls that require it. Generally used with sub-classing to track all Windows messages.

Example:

```

PROGRAM
MAP
  Main
  SubClassFunc(UNSIGNED hWnd,SIGNED wParam,UNSIGNED wParam,LONG lParam),LONG,PASCAL
  MODULE('win32') !SV Windows 32-bit Library
  CallWindowProc(LONG,UNSIGNED,SIGNED,UNSIGNED,LONG),LONG,PASCAL,NAME('CallWindowProcA')
  END
  ENDSavedProc LONG
PT      GROUP,PRE(PT)
X       SHORT
Y       SHORT
      END
CODE
  Main

Main      PROCEDURE
WinView   WINDOW('View'),AT(0,0,320,200),HVSCROLL,MAX,TIMER(1)
  STRING('X Pos'),AT(1,1,,),USE(?String1)
  STRING(@n3),AT(24,1,,),USE(PT:X)
  STRING('Y Pos'),AT(44,1,,),USE(?String2)
  STRING(@n3),AT(68,1,,),USE(PT:Y)
  BUTTON('Close'),AT(240,180,60,20),USE(?Close)
  END
CODE
OPEN(WinView)
SavedProc = WinView{PROP:ClientWndProc}      !Save this procedure
WinView{PROP:ClientWndProc} = ADDRESS(SubClassFunc)
                                           !Change to subclass procedure

ACCEPT
CASE ACCEPTED()
OF ?Close
  BREAK
END
END

```

```

SubClassFunc    PROCEDURE(hWnd,wMsg,wParam,lParam)    !Sub class procedure
WM_MOUSEMOVE    EQUATE(0200H)                        ! to track mouse movement in
CODE                                                    ! client area of window
CASE wMsg
OF WM_MOUSEMOVE
  PT:X = MOUSEX()
  PT:Y = MOUSEY()
END
RETURN(CallWindowProc(SavedProc,hWnd,wMsg,wParam,lParam))
                !Pass control back to
                ! saved procedure

```

PROP:ClipBits

Property of an IMAGE control that allows bitmap images to be moved into (but not out of) the Windows clipboard when set to one (1). Only .BMP, .PCX, or .GIF image types can be stored as a bitmap (.BMP) image in the Clipboard.

Example:

```

WinView    WINDOW('View'),AT(0,0,320,200),MDI,MAX,HVSCROLL
           IMAGE(),AT(0,0,,),USE(?Image)
           BUTTON('Save Picture'),AT(80,180,60,20),USE(?SavePic)
           BUTTON('New Picture'),AT(160,180,60,20),USE(?NewPic)
           END

FileName    STRING(64)                                !Filename variable

CODE
OPEN(WinView)
DISABLE(?LastPic)
IF NOT FILEDIALOG('Choose File to View',FileName,'BitMap|*.BMP|PCX|*.PCX',0)
  RETURN                                           !Return if no file chosen
END
?Image{PROP:Text} = FileName
ACCEPT
CASE ACCEPTED()
OF ?NewPic
  IF NOT FILEDIALOG('Choose File to View',FileName,'BitMap|*.BMP|PCX|*.PCX',0)
    BREAK                                           !Return if no file chosen
  END
  ?Image{PROP:Text} = FileName
OF ?SavePic
  ?Image{PROP:ClipBits} = 1                        !Put image into Clipboard
  ENABLE(?LastPic)                                ! activate Last Picture button
END
END

```

PROP:ColorDialogHook

A property of the SYSTEM built-in variable that sets the override procedure for the COLORDIALOG internal Clarion procedure. Equivalent to {PROP:LibHook,1}. Assign the ADDRESS of the overriding procedure, and the runtime library will call the overriding procedure instead of the COLORDIALOG procedure. Assign zero and the runtime library will once again call its internal procedure. The overriding procedure's prototype must be exactly the same as the COLORDIALOG procedure. (WRITE-ONLY)

PROP:DDEMode

A property of the SYSTEM built-in variable that allows you to set normal DDE event behavior (0, the default) whereby all DDE events are sent to the window that opened the DDE channel, or one (1), which causes all DDE events to be sent to the top window of the current thread.

Example:

```
DDERetVal  STRING(20)
WinOne     WINDOW,AT(0,0,160,400)
           ENTRY(@s20),USE(DDERetVal)
           END
MyServer   LONG
           CODE
           OPEN(WinOne)
           SYSTEM{PROP:DDEMode} = 1           !Send events to top window of current thread
           MyServer = DDESERVER('MyApp','DataEntered') !Open as server
           ACCEPT
           END
```

PROP:DDETimeOut

A property of the SYSTEM built-in variable that allows you to set and get the DDE timeout used for all DDE commands. This value is in hundredths of seconds and the default value is 500.

Example:

```

DDERetVal  STRING(20)
WinOne     WINDOW,AT(0,0,160,400)
           ENTRY(@s20),USE(DDERetVal)
           END
MyServer   LONG
CODE
OPEN(WinOne)
SYSTEM{PROP:DDETimeOut} = 12000           !Set time out to two minutes
MyServer = DDESERVER('MyApp','DataEntered') !Open as server
ACCEPT
CASE EVENT()
OF EVENT:DDErequest           !Data requested once
  DDEWRITE(MyServer,DDE:manual,'DataEntered',DDERetVal)
  !Provide data once
END
END

```

PROP:DeferMove

A property of the SYSTEM built-in variable that defers the resizing and/or movement of controls until the end of the ACCEPT loop or SYSTEM{PROP:DeferMove} is reset to zero (0). This disables the immediate effect of all assignments to position and size properties, enabling the library to perform all moves at once (which eliminates temporarily overlapping controls). The absolute value of the number assigned to SYSTEM{PROP:DeferMove} defines the number of deferred moves for which space is pre-allocated (automatically expanded when necessary, but less efficient and may fail). Assigning a positive number automatically resets PROP:DeferMove to zero at the next ACCEPT, while a negative number leaves it set until explicitly reset to zero (0).

Example:

```

WinView    WINDOW('View'),AT(0,0,320,200),MDI,MAX,HVSCROLL
           IMAGE(),AT(0,0,,),USE(?Image)
           BUTTON('New Picture'),AT(160,180,60,20),USE(?NewPic)
           BUTTON('Close'),AT(80,180,60,20),USE(?Close)
           END
FileName   STRING(64)           !Filename variable
ImageWidth SHORT
ImageHeight SHORT

```

```
CODE
OPEN(WinView)
DISABLE(?LastPic)
IF NOT FILEDIALOG('Choose File to View',FileName,'BitMap|*.BMP|PCX|*.PCX',0)
  RETURN !Return if no file chosen
END
?Image{PROP:Text} = FileName
ACCEPT
CASE ACCEPTED()
OF ?NewPic
  IF NOT FILEDIALOG('Choose File to View',FileName,'BitMap|*.BMP|PCX|*.PCX',0)
    BREAK !Return if no file chosen
  END
  ?Image{PROP:Text} = FileName
  SYSTEM{PROP:DeferMove} = 4 !Defer move and resize
  ImageWidth = ?Image{PROP:Width} !1 move
  ImageHeight = ?Image{PROP:Height} !2 moves
  IF ImageWidth > 320
    ?Image{PROP:Width} = 320
    ?Image{PROP:XPos} = 0
  ELSE
    ?Image{PROP:XPos} = (320 - ImageWidth) / 2 !Center horizontally
  END
  IF ImageHeight > 180
    ?Image{PROP:Height} = 180
    ?Image{PROP:YPos} = 0
  ELSE
    ?Image{PROP:YPos} = (180 - ImageHeight) / 2 !Center vertically
  END
OF ?Close
  BREAK
END !Moves and resizing happen at end of ACCEPT loop
END
```

PROP:Edit

LIST control property which specifies the field equate label of the control to perform edit-in-place for a LIST box column. This is an array whose element number indicates the column number to edit. When non-zero, the control is unhidden and moved/resized over the current row in the column indicated to allow the user to input data. Assign zero to re-hide the data entry control.

Example:

```

Q   QUEUE
f1  STRING(15)
f2  STRING(15)
    END
Win1 WINDOW('List Edit In Place'),AT(0,1,308,172),SYSTEM
    LIST,AT(6,6,120,90),USE(?List),COLUMN,FORMAT('60L@s15@60L@s15@'), |
    FROM(Q),ALRT(EnterKey)
    END
?EditEntry EQUATE(100)
CODE
OPEN(Win1)
CREATE(?EditEntry,CREATE:Entry)
SELECT(?List,1)
ACCEPT
CASE FIELD()
OF ?List
CASE EVENT()
OF EVENT:AlertKey
IF KEYCODE() = EnterKey
SETKEYCODE(MouseLeft2)
POST(EVENT:Accepted,?List)
END
OF EVENT:NewSelection
IF ?List{PROP:edit,?List{PROP:column}}
GET(Q,CHOICE())
END
OF EVENT:Accepted
IF KEYCODE() = MouseLeft2
GET(Q,CHOICE())
?EditEntry{PROP:text} = ?List{PROPLIST:picture,?List{PROP:column}}
CASE ?List{PROP:column}
OF 1
?EditEntry{PROP:use} = Q.F1
OF 2
?EditEntry{PROP:use} = Q.F2
END
?List{PROP:edit,?List{PROP:column}} = ?EditEntry
END
END
END

```

```
OF ?EditEntry
CASE EVENT()
OF EVENT:Selected
  ?EditEntry{PROP:Touched} = 1
OF EVENT:Accepted
  PUT(Q)
  ?List{PROP:edit,?List{PROP:column}} = 0
END
END
END
```

PROP:Enabled

Returns an empty string if the control is not enabled either because it itself has been disabled, or because it is a member of a "parent" control (OPTION, GROUP, MENU, SHEET, or TAB) that has been disabled. (READ-ONLY)

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
SHEET,AT(0,0,320,175),USE(SelectedTab)
TAB('Tab One'),USE(?TabOne)
    PROMPT('Enter Data:'),AT(100,100,20,20),USE(?P1)
    ENTRY(@S8),AT(100,140,32,20),USE(E1)
    PROMPT('Enter More Data:'),AT(100,200,20,20),USE(?P2)
    ENTRY(@S8),AT(100,240,32,20),USE(E2)
    END
TAB('Tab Two'),USE(?TabTwo)
    PROMPT('Enter Data:'),AT(100,100,20,20),USE(?P3)
    ENTRY(@S8),AT(100,140,32,20),USE(E3)
    PROMPT('Enter More Data:'),AT(100,200,20,20),USE(?P4)
    ENTRY(@S8),AT(100,240,32,20),USE(E4)
    END
END
END
BUTTON('Ok'),AT(100,180,20,20),USE(?Ok)
BUTTON('Cancel'),AT(200,180,20,20),USE(?Cancel)
END
CODE
OPEN(MDIChild)
ACCEPT
CASE EVENT()
OF EVENT:Completed
    BREAK
END
CASE FIELD()
OF ?Ok
    CASE EVENT()
    OF EVENT:Accepted
        SELECT
        END
    OF ?E3
        CASE EVENT()
        OF EVENT:Accepted
            IF ?E3{PROP:Enabled} AND MDIChild{PROP:AcceptAll}
                !Check for visibility during AcceptAll mode
            E3 = UPPER(E3)           !Convert the data entered to Upper case
            DISPLAY(?E3)           ! and display the upper cased data
        END
    END
END
```

```
OF ?Cancel
CASE EVENT( )
OF EVENT:Accepted
BREAK
END
END
END
```

PROP:EventsWaiting

WINDOW property which returns whether the window has any further events waiting to be processed. Generally only used by Internet Connect to know when to format an HTML page. (READ-ONLY)

Example:

```
IF TARGET{PROP:EventsWaiting}    !Check for waiting events
  !Do something
END
```

PROP:ExeVersion

A property of the SYSTEM built-in variable that returns the version number of an EXE created by Clarion for Windows. This is the version number of Clarion for Windows which compiled the EXE file, even if the runtime library .DLL is from a newer release (see PROP:LibVersion). This first appeared in Clarion for Windows release 1501, so PROP:ExeVersion returns blank for releases prior to 1501. (READ-ONLY)

Example:

```
MESSAGE('Compiled in CW release ' & SYSTEM{PROP:ExeVersion})
```

PROP:FatalErrorHook

A property of the SYSTEM built-in variable that sets the override procedure for the internal Clarion procedure. Equivalent to {PROP:LibHook,9}. Assign the ADDRESS of the overriding procedure, and the runtime library will call the overriding procedure instead of the internal procedure. Assign zero and the runtime library will once again call its internal procedure. The overriding procedure's prototype must be exactly the same as the internal procedure's (STRING message,UNSIGNED ErrorNumber). (WRITE-ONLY)

PROP:FileDialogHook

A property of the SYSTEM built-in variable that sets the override procedure for the FILEDIALOG internal Clarion procedure. Equivalent to {PROP:LibHook,2}. Assign the ADDRESS of the overriding procedure, and the runtime library will call the overriding procedure instead of the FILEDIALOG procedure. Assign zero and the runtime library will once again call its internal procedure. The overriding procedure's prototype must be exactly the same as the FILEDIALOG procedure. (WRITE-ONLY)

PROP:FlushPageNumFunc

A property of a REPORT that sets the callback function called during the flushing of previewed pages. This callback function returns the number of the next page to be flushed. The callback function must be a PROCEDURE that takes a single LONG parameter and returns a LONG.

To turn this on, the ADDRESS of your callback function must be assigned to PROP:FlushPageNumFunc. To turn it off, assign zero (0) to it.

When the report engine flushes report pages to a printer as a result of the execution of Report{PROP:FlushPreview}, the callback function is called to retrieve the number of the next page to be printed. The parameter passed to the callback function is the number of the last printed page, or 0 before printing of the first page. If the callback function returns a value out of the range of pages in the report, the report engine stops printing and removes all records from the preview queue.

The PROP:FlushPageNumFunc can return the same page number multiple times prior to the closing of the report. Once the report is closed the report engine removes the created metafiles.

Example:

```
MAP
  PageNum (LONG), LONG
END

CODE
OPEN (Report)
!Report processing code here
Report{PROP:FlushPageNumFunc} = ADDRESS(PageNum)
Report{PROP:FlushPreview} = TRUE
Report{PROP:FlushPageNumFunc} = 0

PageNum PROCEDURE (LONG LastPage)
CODE
IF LastPage = 0
  RETURN 1
ELSE
  RETURN LastPage + 2 ! Print odd pages only
END
```

PROP:FlushPreview

Flushes the REPORT structure's PREVIEW attribute metafiles to the printer (0 = off, else on, always 0 at report open).

Example:

```
SomeReport PROCEDURE
WMFQue  QUEUE           !Queue to contain .WMF filenames
        STRING(64)
        END
NextEntry BYTE(1)       !Queue entry counter variable

Report  REPORT,PREVIEW(WMFQue) !Report with PREVIEW attribute
DetailOne  DETAIL
          !Report controls
          END
          END

ViewReport WINDOW('View Report'),AT(0,0,320,200),MDI,MAX,HVSCROLL
          IMAGE(),AT(0,0,320,180),USE(?ImageField)
          BUTTON('View Next Page'),AT(0,180,60,20),USE(?NextPage),DEFAULT
          BUTTON('Print Report'),AT(80,180,60,20),USE(?PrintReport)
          BUTTON('Exit Without Printing'),AT(160,180,60,20),USE(?ExitReport)
          END

CODE
OPEN(Report)
SET(SomeFile)           !Code to generate the report
LOOP
  NEXT(SomeFile)
  IF ERRORCODE() THEN BREAK.
  PRINT(DetailOne)
END
ENDPAGE(Report)
OPEN(ViewReport)       !Open report preview window
GET(WMFQue,NextEntry)  !Get first queue entry
?ImageField{PROP:text} = WMFQue !Load first report page
ACCEPT
CASE ACCEPTED()
OF ?NextPage
  NextEntry += 1       !Increment entry counter
  IF NextEntry > RECORDS(WMFQue) THEN CYCLE. !Check for end of report
  GET(WMFQue,NextEntry) !Get next queue entry
  ?ImageField{PROP:text} = WMFQue !Load next report page
  DISPLAY                ! and display it
OF ?PrintReport
  Report{PROP:FlushPreview} = 1 !Flush files to printer
```

```

BREAK                ! and exit procedure
OF ?ExitReport
BREAK                !Exit procedure
END
END
RETURN               !Return to caller, automatically
                    ! closing the window and report
                    ! freeing the queue and automatically
                    ! deleting all the temporary .WMF files

```

PROP:Follows

Changes the tab order to specify the position within the parent that the control will occupy. The control follows the control number you specify in the tab order. This must specify an existing control within the parent (window, option, group, menu, report, detail, etc.). Setting PROP:Follows to a REGION control will be ignored, as REGIONS are not in the Windows tab order. (WRITE-ONLY)

Example:

```

WinView  WINDOW('View Report'),AT(0,0,320,200),MDI,MAX,HVSCROLL
         BUTTON('View Next Page'),AT(0,180,60,20),USE(?NextPage),DEFAULT
         BUTTON('Print Report'),AT(80,180,60,20),USE(?PrintReport)
         BUTTON('Exit Without Printing'),AT(160,180,60,20),USE(?ExitReport)
         END
CODE
OPEN(WinView)
        !Print Report button normally follows View button
?PrintReport{PROP:Follows} = ?ExitReport
        !Now Print Report button follows Exit button in the tab order
ACCEPT
END

```

PROP:FontDialogHook

A property of the SYSTEM built-in variable that sets the override procedure for the FONTDIALOG internal Clarion procedure. Equivalent to {PROP:LibHook,3}. Assign the ADDRESS of the overriding procedure, and the runtime library will call the overriding procedure instead of the FONTDIALOGA procedure. Assign zero and the runtime library will once again call its internal procedure. The overriding procedure's prototype must be exactly the same as the FONTDIALOGA procedure. (WRITE-ONLY)

PROP:HaltHook

A property of the SYSTEM built-in variable that sets the override procedure for the HALT internal Clarion procedure. Equivalent to {PROP:LibHook,5}. Assign the ADDRESS of the overriding procedure, and the runtime library will call the overriding procedure instead of the HALT procedure. Assign zero and the runtime library will once again call its internal procedure. The overriding procedure's prototype must be exactly the same as the HALT procedure. (WRITE-ONLY)

PROP:Handle

Returns the window or control handle for use with low-level Windows API calls that require it. It is also used when assigning the handle of one BLOB to another.

Example:

```
WinView  WINDOW('View'),AT(0,0,320,200),MDI,MAX,HVSCROLL
        END
MessageText  CSTRING('You cannot exit the program from this window ')
MessageCaption CSTRING('No EVENT:CloseDown Allowed ')
TextAddress  LONG
CaptionAddress LONG
RetVal      SHORT
CODE
OPEN(WinView)
ACCEPT
CASE EVENT()
OF EVENT:CloseDown
    TextAddress = ADDRESS(MessageText)
    CaptionAddress = ADDRESS(MessageCaption)
    RetVal = MessageBox(WinView{PROP:Handle},TextAddress,CaptionAddress,MB_OK)
                !Windows API call using a window handle
CYCLE
                !Disallow program closedown from this window
END
END
```

PROP:HeaderHeight

Returns the height of the header in a LIST or COMBO control. The height is measured in dialog units (unless PROP:Pixels is active). (READ-ONLY)

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
    LIST,AT(0,0,220,220),USE(?L1),FROM(Que),IMM,FORMAT('60L~Header Text~')
    END
CODE
OPEN(MDIChild)
X# = ?L1{PROP:HeaderHeight}      !Get height of header in dialog units
```

PROP:HscrollPos

Returns the position of the horizontal scroll bar's "thumb" (from 0 to 255) on a window, IMAGE, TEXT, LIST or COMBO with the HSCROLL attribute. Setting this causes the control or window's contents to scroll horizontally.

Example:

```

Que      QUEUE
F1       STRING(50)
F2       STRING(50)
F3       STRING(50)
        END
WinView  WINDOW('View'),AT(,,340,200),SYSTEM,CENTER
        LIST,AT(20,0,300,200),USE(?List),FROM(Que),IMM,HVSCROLL |
        FORMAT('80L#1#80L#2#80L#3#')
        END
CODE
OPEN(WinView)
DO BuildListQue
ACCEPT
CASE FIELD()
OF ?List
CASE EVENT()
OF EVENT:ScrollDrag
CASE (?List{PROP:HscrollPos} % 200) + 1
OF 1
?List{PROP:Format} = '80L#1#80L#2#80L#3#'
OF 2
?List{PROP:Format} = '80L#2#80L#3#80L#1#'
OF 3
?List{PROP:Format} = '80L#3#80L#1#80L#2#'
END
DISPLAY
. . .
FREE(Que)
BuildListQue ROUTINE
LOOP 15 TIMES
Que.F1 = 'F1F1F1F1'
Que.F2 = 'F2F2F2F2'
Que.F3 = 'F3F3F3F3'
ADD(Que)
END

```

PROP:IconList

An array that sets or returns the icons displayed in a LIST formatted to display icons (usually a tree control). If the name of the icon file to assign has a number in square brackets appended to its end, this indicates the file contains multiple icons and the number specifies which to assign (zero-based). If the name of the icon file has a tilde (~) prepended to it (~IconFile.ICO), this indicates the file has been linked into the project as a resource and is not on disk.

Example:

```

PROGRAM
MAP
RandomAlphaData PROCEDURE(*STRING Field)
    END
TreeDemo    QUEUE,PRE()      !Data list box FROM queue
FName       STRING(20)
ColorNFG    LONG             !Normal Foreground color for FName
ColorNBG    LONG             !Normal Background color for FName
ColorSFG    LONG             !Selected Foreground color for FName
ColorSBG    LONG             !Selected Background color for FName
IconField   LONG             !Icon number for FName
TreeLeve    LONG             !Tree Level
LName       STRING(20)
Init        STRING(4)
            END

Win WINDOW('List Boxes'),AT(0,0,366,181),SYSTEM,DOUBLE
    LIST,AT(0,34,366,146),FROM(TreeDemo),USE(?Show),HVSCROLL, |
        FORMAT('80L*IT~First Name~*80L~Last Name~16C~Initials~')
    END
CODE
LOOP 20 TIMES
    RandomAlphaData(FName)
    ColorNFG = COLOR:White           !Assign FNAME's colors
    ColorNBG = COLOR:Maroon
    ColorSFG = COLOR:Yellow
    ColorSBG = COLOR:Blue
    IconField = ((x#-1) % 4) + 1     !Assign icon number
    TreeLevel = ((x#-1) % 4) + 1    !Assign tree level
    RandomAlphaData(LName)
    RandomAlphaData(Init)
    ADD(TD)
END
OPEN(Win)
?Show{PROP:iconlist,1} = ICON:VCRback      !Icon 1 = <
?Show{PROP:iconlist,2} = ICON:VCRrewind    !Icon 2 = <<
?Show{PROP:iconlist,3} = 'VCRdown.ico'     !Icon 3 = > not linked into project
?Show{PROP:iconlist,4} = '~VCRnext.ico'    !Icon 4 = >>linked into project
ACCEPT

```

END

RandomAlphaData PROCEDURE(*STRING Field)

CODE

y# = RANDOM(1,SIZE(Field)) !Random fill size

LOOP x# = 1 to y# !Fill each character with

 Field[x#] = CHR(RANDOM(97,122)) ! a random lower case letter

END

PROP:ImageBits

Property of an IMAGE control that allows bitmap images displayed in the control to be moved into and out of memo fields. Any image displayed in the control can be stored. PROP:ImageBlob performs the same type of function for a BLOB.

Example:

```

WinView  WINDOW('View'),AT(0,0,320,200),MDI,MAX,HVSCROLL
         IMAGE(),AT(0,0,,),USE(?Image)
         BUTTON('Save Picture'),AT(80,180,60,20),USE(?SavePic)
         BUTTON('New Picture'),AT(160,180,60,20),USE(?NewPic)
         BUTTON('Last Picture'),AT(240,180,60,20),USE(?LastPic)
END

SomeFile  FILE,DRIVER('Clarion'),PRE(Fil)      !A file with a memo field
MyMemo    MEMO(65520),BINARY
Rec       RECORD
Fl        LONG
. .

FileName  STRING(64)                          !Filename variable

CODE
OPEN(SomeFile)
OPEN(WinView)
DISABLE(?LastPic)
IF NOT FILEDIALOG('Choose File to View',FileName,'BitMap|*.BMP|PCX|*.PCX',0)
  RETURN                                     !Return if no file chosen
END
?Image{PROP:Text} = FileName
ACCEPT
CASE ACCEPTED()
OF ?NewPic
  IF NOT FILEDIALOG('Choose File to View',FileName,'BitMap|*.BMP|PCX|*.PCX',0)
    BREAK                                   !Return if no file chosen
  END
  ?Image{PROP:Text} = FileName
OF ?SavePic
  Fil:MyMemo = ?Image{PROP:ImageBits}      !Put image into memo
  ADD(SomeFile)                            ! and save it to the file on disk
  ENABLE(?LastPic)                         ! activate Last Picture button
OF ?LastPic
  ?Image{PROP:ImageBits} = Fil:MyMemo      !Put last saved memo into image
END
END
END

```

PROP:ImageBlob

Property of an IMAGE control that allows bitmap images displayed in the control to be moved into and out of BLOB fields. Any image displayed in the control can be stored. PROP:ImageBits performs the same type of function for a MEMO. Most images are stored in a bitmap format by default (except PCX and GIF), unless PROP:PrintMode is set to store the native format.

PROP:InitAStringHook

A property of the SYSTEM built-in variable that allows a procedure to be set up to automatically initialize the TIE values the first time an ASTRING representing a particular value is assigned. Equivalent to {PROP:LibHook,14}. Assign the ADDRESS of the InitAString procedure, and the runtime library will call this procedure when an ASTRING variable is first assigned. Assign zero and the runtime library will stop calling the InitAString procedure. (WRITE-ONLY)

PROP:Interface

A property of a TEXT control used to get an interface to the RTF control's host interface.

PROP:InToolbar

A toggle attribute which returns whether the control is in a TOOLBAR structure. (READ-ONLY)

Example:

```
WinView WINDOW('View'),AT(0,0,,),MDI,MAX,HVSCROLL,SYSTEM,RESIZE
  TOOLBAR
    BUTTON('Save Picture'),AT(80,180,60,20),USE(?SavePic)
  END
  LIST,AT(6,6,120,90),USE(?List),FORMAT('120L'),FROM(Q),IMM
END

CODE
OPEN(WinView)
IF ?SavePic{PROP:InToolbar} = TRUE
  !DO Something
END
ACCEPT
END
```

PROP:Items

Returns or sets the number of entries visible in a LIST or COMBO control.

Example:

```

Que      QUEUE
        STRING(30)
        END

WinView  WINDOW('View'),AT(0,0,320,200),MDI,MAX,HVSCROLL,SYSTEM
        LIST,AT(6,6,120,90),USE(?List),FORMAT('120L'),FROM(Q),IMM
        END

CODE
OPEN(WinView)
SET(SomeFile)
LOOP ?List{PROP:Items} TIMES !Fill display queue to limit of displayable
items
    NEXT(SomeFile)
    Que = Fil:Record
    ADD(Que)
END
ACCEPT
END

```

PROP:LazyDisplay

SYSTEM property which disables (when set to 1) or enables (when set to 0, the default) the feature where all window re-painting is completely done before processing continues with the next statement following a DISPLAY. Setting PROP:LazyDisplay = 1 creates seemingly faster video processing, since the re-paints occur at the end of the ACCEPT loop if there are no other messages pending. This can improve the performance of some applications, but can also have a negative impact on appearance.

Example:

```

WinView  APPLICATION('MyApp'),AT(0,0,320,200),MAX,HVSCROLL,SYSTEM
        END

CODE
OPEN(WinView)
SYSTEM{PROP:LazyDisplay} = 1 !Disable extra paint message display
! throughout entire application
ACCEPT
END

```

PROP:LFNSupport

NOTE: This property has been deprecated in this release.

A property of the SYSTEM built-in variable in 16-bit programs that returns one (1) if the operating system supports long filenames, and an empty string (") if it does not. 32-bit operating systems all support long filenames. (READ-ONLY)

Example:

```
IF SYSTEM{PROP:LFNSupport} = TRUE
  MESSAGE('Long Filenames are supported')
ELSE
  MESSAGE('Long Filenames are NOT supported')
END
```

PROP:LibHook

An array property of the SYSTEM built-in variable that sets override procedures for several internal Clarion procedures. For each of these procedures, you assign the ADDRESS of the overriding procedure, and the runtime library will call the overriding procedure instead of the Clarion library procedure. The overriding procedure's prototype must be exactly the same as the internal Clarion procedure. These properties were implemented to facilitate Internet Connect. (WRITE-ONLY)

```
{PROP:Libhook, 1} PROP:ColorDialogHook
{PROP:Libhook, 2} PROP:FileDialogHook
{PROP:Libhook, 3} PROP:FontDialogHook
{PROP:Libhook, 4} PROP:PrinterDialogHook
{PROP:Libhook, 5} PROP:HaltHook
{PROP:Libhook, 6} PROP:MessageHook
{PROP:Libhook, 7} PROP:StopHook
{PROP:Libhook, 8} PROP:AssertHook
{PROP:Libhook, 9} PROP:FatalErrorHook
{PROP:Libhook, 12} PROP:SystemPropHook
{PROP:Libhook, 13} PROP:AssertHook2
{PROP:Libhook, 14} PROP:InitAStringHook
{PROP:Libhook, 15} PROP:UnlockThreadHook
{PROP:Libhook, 16} PROP:LockThreadHook
{PROP:Libhook, 17} PROP:ThreadLockedHook
```

PROP:LibVersion

A property of the SYSTEM built-in variable that returns the version number of the Clarion for Windows runtime library .DLL currently loaded for the EXE currently executing. This is separate from the version number of Clarion for Windows which compiled the EXE file (see PROP:ExeVersion). This first appeared in Clarion for Windows release 1501, so PROP:ExeVersion returns blank for releases prior to 1501. (READ-ONLY)

Example:

```
MESSAGE('Runtime DLL from release ' & SYSTEM{PROP:LibVersion})
```

PROP:Line and PROP:LineCount

PROP:Line is an array whose elements each contain one line of the text in a TEXT control. (READ ONLY)

PROP:LineCount returns the number of lines of text in a TEXT control. (READ ONLY)

Example:

```
LineCount SHORT
MemoLine STRING(80)

CustRpt  REPORT,AT(1000,1000,6500,9000),THOUS
Detail1  DETAIL,AT(0,0,6500,6000)
         TEXT,AT(0,0,6500,6000),USE(Fil:MemoField)
         END
Detail2  DETAIL,AT(0,0,6500,125)
         STRING(@s80),AT(0,0,6500,125),USE(MemoLine)
         END
         END

CODE
OPEN(File)
SET(File)
OPEN(CustRpt)
LOOP
  NEXT(File)
  LineCount = CustRpt$?Fil:MemoField{PROP:LineCount}
  LOOP X# = 1 TO LineCount
    MemoLine = CustRpt$?Fil:MemoField{PROP:Line,X#}
    PRINT(Detail2)
  END
END
```

PROP:LineHeight

Sets or returns the height of the rows in a LIST or COMBO control. The height is measured in dialog units (unless PROP:Pixels is active). For a TEXT control, it returns the character cell height of the control's font (the distance from the top of one line of text to the top of the next) in whatever measurement unit is currently in use. READ-ONLY for a TEXT control.

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
    LIST,AT(0,0,220,220),USE(?L1),FROM(Que),IMM
    END
CODE
OPEN(MDIChild)
?L1{PROP:LineHeight} = 8           !Set height to 8 dialog units
```

PROP:MaxHeight

PROP:MaxHeight sets or returns the maximum height of a resizable window.

Example:

```
WinView WINDOW('View'),AT(0,0,320,200),MDI,MAX,HVSCROLL,SYSTEM,RESIZE
    LIST,AT(6,6,120,90),USE(?List),FORMAT('120L'),FROM(Q),IMM
    END

CODE
OPEN(WinView)
WinView{PROPMaxHeight} = 200      !Set boundaries beyond which the user cannot
WinView{PROPMaxWidth} = 320      ! resize the window
WinView{PROPMinHeight} = 90
WinView{PROPMinWidth} = 120
ACCEPT
END
```

PROP:MaxWidth

PROP:MaxWidth sets or returns the maximum width of a resizable window.

Example:

```
WinView WINDOW('View'),AT(0,0,320,200),MDI,MAX,HVSCROLL,SYSTEM,RESIZE
        LIST,AT(6,6,120,90),USE(?List),FORMAT('120L'),FROM(Q),IMM
        END

CODE
OPEN(WinView)
WinView{PROPMaxHeight} = 200      !Set boundaries beyond which the user cannot
WinView{PROPMaxWidth} = 320      ! resize the window
WinView{PROPMinHeight} = 90
WinView{PROPMinWidth} = 120
ACCEPT
END
```

PROP:MessageHook

A property of the SYSTEM built-in variable that sets the override procedure for the MESSAGE internal Clarion procedure. Equivalent to {PROP:LibHook,6}. Assign the ADDRESS of the overriding procedure, and the runtime library will call the overriding procedure instead of the MESSAGE procedure. Assign zero and the runtime library will once again call its internal procedure. The overriding procedure's prototype must be exactly the same as the MESSAGE procedure. (WRITE-ONLY)

PROP:MinHeight

PROP:MinHeight sets or returns the minimum height of a resizable window.

Example:

```
WinView WINDOW('View'),AT(0,0,320,200),MDI,MAX,HVSCROLL,SYSTEM,RESIZE
      LIST,AT(6,6,120,90),USE(?List),FORMAT('120L'),FROM(Q),IMM
      END

CODE
OPEN(WinView)
WinView{PROPMaxHeight} = 200      !Set boundaries beyond which the user cannot
WinView{PROPMaxWidth} = 320      ! resize the window
WinView{PROPMinHeight} = 90
WinView{PROPMinWidth} = 120
ACCEPT
END
```

PROP:MinWidth

PROP:MinWidth sets or returns the minimum width of a resizable window. Also sets the minimum width of TAB controls in a SHEET.

Example:

```
WinView WINDOW('View'),AT(0,0,320,200),MDI,MAX,HVSCROLL,SYSTEM,RESIZE|
      LIST,AT(6,6,120,90),USE(?List),FORMAT('120L'),FROM(Q),IMM
      END

CODE
OPEN(WinView)
WinView{PROPMaxHeight} = 200      !Set boundaries beyond which the user cannot
WinView{PROPMaxWidth} = 320      ! resize the window
WinView{PROPMinHeight} = 90
WinView{PROPMinWidth} = 120
ACCEPT
END
```

PROP:NextField

An array property which returns the next control number in sequence of a window or report. (READ-ONLY) The control number returned is the control following the array element numbered control. The order in which PROP:NextField returns field numbers is undefined. PROP:NextField returns zero when the array element number is the last control in its list. This property easily allows you to loop through all the controls in a window or report, whether those controls have USE attributes or not.

Example:

```
WinView WINDOW('View'),AT(0,0,320,200),MDI,MAX,HVSCROLL
    IMAGE(),AT(0,0,,),USE(?Image)
    BUTTON('Save Picture'),AT(80,180,60,20),USE(?SavePic)
    BUTTON('New Picture'),AT(160,180,60,20),USE(?NewPic)
    BUTTON('Last Picture'),AT(240,180,60,20),USE(?LastPic)
END
ThisField SHORT(0)
CODE
OPEN(WinView)
LOOP
    ThisField = WinView{PROP:NextField,ThisField} !Process every control
    IF ThisField
        ThisField{PROP:FontName} = 'Arial' !Changing the font
        ThisField{PROP:FontSize} = 10
    ELSE
        BREAK !Break when done
    . .
ACCEPT
END
```

PROP:NextPageNo

A property which sets or returns the next page number in a report.

Example:

```

CustRpt  REPORT,AT(1000,1000,6500,9000),THOUS
        HEADER
            STRING(@n3),USE(?Page),PAGE NO
        END
Detail   DETAIL,AT(0,0,6500,1000)
        STRING,AT(10,10),USE(Fil:Field)
        .
        .
CODE
OPEN(File);SET(File)
OPEN(CustRpt)
LOOP
NEXT(File)
IF Fil:KeyField <> Sav:KeyField !Detect group break
Sav:KeyField = Fil:KeyField !Detect group break
ENDPAGE !Force page break
CustRpt{PROP:NextPageNo} = 1 !Every group starts on page one
END
PRINT(Detail)
END

```

PROP:NextTabStop and PROP:PrevTabStop

PROP:NextTabStop returns the next control (field equate label) that can have input focus and be reached by pressing the Tab key.

PROP:PrevTabStop returns the previous control (field equate label) that can have input focus and be reached by pressing ShiftTab key

Example:

```

NextControlFEQ = window{PROP:NextTabStop} !get next control to TAB to
PrevControlFEQ = window{PROP:PrevTabStop} !Get next control to shift-tab to
Window{PROP:NextTabStop} = ?List !Set the next control to tab to

```

PROP:NoHeight and PROP:NoWidth

PROP:NoHeight is a toggle attribute which returns whether the window or control was set to default its height (had an omitted height parameter in its AT attribute). Setting this property to TRUE is equivalent to resetting the control to its default height as determined by the library (which you cannot do using PROP:Height).

PROP:NoWidth is a toggle attribute which returns whether the window or control was set to default its width (had an omitted width parameter in its AT attribute). Setting this property to TRUE is equivalent to resetting the control to its default width as determined by the library (which you cannot do using PROP:Width).

Example:

```
WinView WINDOW('View'),AT(0,0,,),MDI,MAX,HVSCROLL,SYSTEM,RESIZE
        LIST,AT(6,6,120,90),USE(?List),FORMAT('120L'),FROM(Q),IMM
        END
```

```
CODE
OPEN(WinView)
IF WinView{PROP:NoHeight} = TRUE
    WinView{PROP:Height} = 200           !Set height
END
IF WinView{PROP:NoWidth} = TRUE
    WinView{PROP:Width} = 320          !Set width
END
ACCEPT
END
```

PROP:NoTips

Disables (when set to 1) or re-enables (when set to 0) tooltip display (TIP attribute) for the SYSTEM, window, or control.

Example:

```
WinView  APPLICATION('MyApp'),AT(0,0,320,200),MAX,HVSCROLL,SYSTEM
        END
CODE
OPEN(WinView)
SYSTEM{PROP:NoTips} = 1  !Disable TIP display throughout entire application
ACCEPT
END
```

PROP:NumTabs

PROP:NumTabs returns the number of TABs in a SHEET. (READ-ONLY)

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
SHEET,AT(0,0,320,175),USE(SelectedTab)
TAB('Tab One'),USE(?TabOne)
    OPTION('Option 1'),USE(OptVar1),KEY(F10Key),HLP('Option1Help')
    RADIO('Radio 1'),AT(20,0,20,20),USE(?R1)
    RADIO('Radio 2'),AT(40,0,20,20),USE(?R2)
END
OPTION('Option 2'),USE(OptVar2),MSG('Option 2')
RADIO('Radio 3'),AT(60,0,20,20),USE(?R3)
RADIO('Radio 4'),AT(80,0,20,20),USE(?R4)
END
END
TAB('Tab Two'),USE(?TabTwo)
    PROMPT('Enter Data:'),AT(100,100,20,20),USE(?P1)
    ENTRY(@S8),AT(100,140,32,20),USE(E1)
    PROMPT('Enter More Data:'),AT(100,200,20,20),USE(?P2)
    ENTRY(@S8),AT(100,240,32,20),USE(E2)
END
TAB('Tab Three'),USE(?TabThree)
    OPTION('Option 3'),USE(OptVar3)
    RADIO('Radio 1'),AT(20,0,20,20),USE(?R5)
    RADIO('Radio 2'),AT(40,0,20,20),USE(?R6)
END
OPTION('Option 4'),USE(OptVar4)
RADIO('Radio 3'),AT(60,0,20,20),USE(?R7)
RADIO('Radio 4'),AT(80,0,20,20),USE(?R8)
END
END
TAB('Tab Four'),USE(?TabFour)
    PROMPT('Enter Data:'),AT(100,100,20,20),USE(?P3)
    ENTRY(@S8),AT(100,140,32,20),USE(E3)
    PROMPT('Enter More Data:'),AT(100,200,20,20),USE(?P4)
    ENTRY(@S8),AT(100,240,32,20),USE(E4)
END
END
BUTTON('Ok'),AT(100,180,20,20),USE(?Ok)
BUTTON('Cancel'),AT(200,180,20,20),USE(?Cancel)
END
CODE
OPEN(MDIChild)
MESSAGE('Number of TABs: ' & ?SelectedTab{PROP:NumTabs})
MESSAGE('Number of rows of TABs: ' & ?SelectedTab{PROP:TabRows})
ACCEPT;END
```

PROP:Parent

Returns the parent control for a control within a structure (such as an OPTION or GROUP control structure, or a DETAIL, TOOLBAR, or MENUBAR structure). (READ ONLY)

Example:

```
WinView WINDOW('MyApp'),AT(0,0,320,200),MAX,HVSCROLL,SYSTEM
    END
OptionSelected STRING(6)
?OptionControl EQUATE(100)      !A field equate number for CREATE to use
?Radio1 EQUATE(101)             !A field equate number for CREATE to use
?Radio2 EQUATE(102)             !A field equate number for CREATE to use
CODE
OPEN(WinView)
CREATE(?OptionControl,CREATE:option)      !Create the OPTION control
?OptionControl{PROP:use} = OptionSelected
?OptionControl{PROP:Text} = 'Pick one'
?OptionControl{PROP:Boxed} = TRUE
SETPOSITION(?OptionControl,10,10)
CREATE(?Radio1,CREATE:radio,?OptionControl)      !Create a RADIO control
?Radio1{PROP:Text} = 'First'
SETPOSITION(?Radio1,12,20)
CREATE(?Radio2,CREATE:radio,?Radio1{PROP:Parent}) !Create another with same
parent
?Radio2{PROP:Text} = 'Second'
SETPOSITION(?Radio2,12,30)
UNHIDE(?OptionControl,?Radio2)              !Display the new controls
ACCEPT
END
```

PROP:Pixels

WINDOW property which toggles screen measurement between dialog units (DLUs) and pixels (not available for reports). After setting this property, all screen positioning (such as GETPOSITION, SETPOSITION, MOUSEX, MOUSEY, PROP:Xpos, PROP:Ypos, PROP:Width, and PROP:Height) return and require co-ordinates in pixels rather than DLUs.

Example:

```
WinView WINDOW('MyApp'),AT(0,0,320,200),MAX,HVSCROLL,SYSTEM
    END
CODE
OPEN(WinView)
WinView{PROP:Pixels} = 1 !Change measurement to pixels
ACCEPT
!all screen positioning statements now return pixels
END
```

PROP:PrinterDialogHook

A property of the SYSTEM built-in variable that sets the override procedure for the PRINTERDIALOG internal Clarion procedure. Equivalent to {PROP:LibHook,4}. Assign the ADDRESS of the overriding procedure, and the runtime library will call the overriding procedure instead of the PRINTERDIALOG procedure. Assign zero and the runtime library will once again call its internal procedure. The overriding procedure's prototype must be exactly the same as the PRINTERDIALOG procedure. (WRITE-ONLY)

PROP:PrintMode

Bitmap property of an IMAGE control (or SYSTEM) that specifies how PROP:ImageBlob stores images in the BLOB. Bit 0 indicates whether decoded image information is required, and bit 1 indicates whether undecoded image information is required. When set to 3, both the original data and decoded DIB data are available, allowing PROP:ImageBlob to store the image in its native format (such as JPG) in the BLOB.

Example:

```
WinView  WINDOW('View'),AT(0,0,320,200),MDI,MAX,HVSCROLL
          IMAGE(),AT(0,0,,),USE(?Image)
          BUTTON('Save Picture'),AT(80,180,60,20),USE(?SavePic)
          BUTTON('New Picture'),AT(160,180,60,20),USE(?NewPic)
          BUTTON('Last Picture'),AT(240,180,60,20),USE(?LastPic)
          END
SomeFile  FILE,DRIVER('TopSpeed'),PRE(Fil)    !A file with a memo field
MyBlob    BLOB,BINARY
Rec        RECORD
F1         LONG
.
.
FileName  STRING(64)                          !Filename variable
CODE
OPEN(SomeFile); OPEN(WinView)
DISABLE(?LastPic)
IF NOT FILEDIALOG('File to View',FileName,'Images|*.BMP;*.PCX;*.JPG;*.WMF',0)
  RETURN                                       !Return if no file chosen
END
?Image{PROP:PrintMode} = 3                   !Setup to store native formats
?Image{PROP:Text} = FileName
ACCEPT
CASE ACCEPTED()
OF ?NewPic
  IF NOT FILEDIALOG('File to
View',FileName,'Images|*.BMP;*.PCX;*.JPG;*.WMF',0)
    BREAK
  END
  ?Image{PROP:Text} = FileName
```

```

OF ?SavePic
  Fil:MyBlob{PROP:Handle} = ?Image{PROP:ImageBlob}    !Put image into BLOB
  ADD(SomeFile)                ! and save it to the file on disk
  ENABLE(?LastPic)             ! activate Last Picture button
OF ?LastPic
  ?Image{PROP:ImageBlob} = Fil:MyBlob{PROP:Handle}    !Put last saved BLOB
into image
  END
END

```

PROP:Progress

You can directly update the display of a PROGRESS control by assigning a value (which must be within the range defined by the RANGE attribute) to the control's PROP:progress property.

Example:

```

BackgroundProcess PROCEDURE          !Background processing batch process

Win WINDOW('Batch Processing...'),AT(,,400,400),TIMER(1),MDI,CENTER
  PROGRESS,AT(100,140,200,20),USE(?ProgressBar),RANGE(0,200)
  BUTTON('Cancel'),AT(190,300,20,20),STD(STD:Close)
  END

CODE
OPEN(Win)
OPEN(File)
?ProgressBar{PROP:rangehigh} = RECORDS(File)
SET(File)                !Set up a batch process
ACCEPT
CASE EVENT()
OF EVENT:CloseWindow
  BREAK
OF EVENT:Timer           !Process records when timer allows it
  ProgressVariable += 3    !Auto-updates 1st progress bar
  LOOP 3 TIMES
  NEXT(File)
  IF ERRORCODE() THEN BREAK.
  ?ProgressBar{PROP:progress} = ?ProgressBar{PROP:progress} + 1
    !Manually update progress bar
  !Perform some batch processing code
  END
END
END
CLOSE(File)

```

PROP:PropVscroll

This property is used to determine how the vertical scroll bar's "thumb." Is displayed.

If set to 1 and the number of records in the list box source (normally a queue) is greater than number of list box visible rows, the thickness of the "thumb" is dependent on the ratio of number of visible rows to the total number of records.

If set to 0, or the number of records is not greater than number of rows (i.e., page loaded browses), the "thumb" has a square shape.

This property is applicable for LIST and COMBO controls and for SYSTEM. New LIST and COMBO controls inherit the current value of this property from the SYSTEM setting.

PROP:RejectCode

ENTRY, TEXT, COMBO, or SPIN control property which returns the last value REJECTCODE value set in EVENT:Rejected. PROP:RejectCode is persistent, while the REJECTCODE procedure only returns a valid value during EVENT:Rejected.

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
  PROMPT('Enter Data:'),AT(100,100,20,20),USE(?P1)
  ENTRY(@N8),AT(100,140,32,20),USE(E1)
  BUTTON('Ok'),AT(100,180,20,20),USE(?Ok)
  BUTTON('Cancel'),AT(200,180,20,20),USE(?Cancel)
END

CODE
OPEN(MDIChild)
ACCEPT
CASE EVENT()
OF EVENT:Completed
  BREAK
END
CASE FIELD()
OF ?Ok
  CASE EVENT()
  OF EVENT:Accepted
    SELECT
  END
OF ?E1
  CASE EVENT()
  OF EVENT:Accepted
    IF ?E1{PROP:RejectCode} <> 0 !Check for rejected entry
      SELECT(?) ! and make the user re-enter
      CYCLE ! immediately
    END
  END
OF ?Cancel
  CASE EVENT()
  OF EVENT:Accepted
    BREAK
  END
END
END
```

PROP:ScreenText

Returns the text displayed on screen in the specified control.

Example:

```
WinView  WINDOW('View'),AT(0,0,320,200),MDI,MAX,HVSCROLL
         SPIN(@n3),AT(0,0,320,180),USE(Fil:Field),RANGE(0,255)
         END
CODE
OPEN(WinView)
ACCEPT
CASE FIELD()
OF ?Fil:Field
CASE EVENT()
OF EVENT:Rejected
MESSAGE(?Fil:Field{PROP:ScreenText} & ' is not in the range 0-255')
SELECT(?)
CYCLE
END
END
END
```

PROP:SelStart (or PROP:Selected) and PROP:SelEnd

PROP:SelStart (also named PROP:Selected) sets or retrieves the beginning (inclusive) character to mark as a block in an ENTRY or TEXT control. It positions the data entry cursor left of the character, and sets PROP:SelEnd to zero (0) to indicate no block is marked. It also identifies the currently highlighted entry in a LIST or COMBO control (usually coded as PROP:Selected for this purpose).

PROP:SelEnd sets or retrieves the ending (inclusive) character to mark as a block in an ENTRY or TEXT control.

Example:

```
WinView  WINDOW('View'),AT(0,0,320,200),MDI,MAX,HVSCROLL
          ENTRY(@S30),AT(0,0,320,180),USE(Fil:Field),ALRT(F10Key)
          LIST,AT(6,6,120,90),USE(?List),FORMAT('120L'),FROM(Q),IMM
          END
CODE
OPEN(WinView)
ACCEPT
CASE ACCEPTED()
OF ?List
  GET(Q,?List{PROP:Selected})      !Get highlighted entry from queue
OF ?Fil:Field
  SETCLIPBOARD(Fil:Field[?Fil:Field{PROP:SelStart} : ?Fil:Field{PROP:SelEnd}])
                                !Place highlighted string slice in Windows' clipboard
END
END
```

PROP:Size

Returns or sets the size of a BLOB field. Before assigning data to the BLOB using the string slicing technique, if the BLOB does not yet contain any data you must set its size using PROP:Size. Before assigning additional data that will increase the amount of data in the BLOB (using the string slicing technique), you must reset its size using PROP:Size.

Example:

```
Names    FILE,DRIVER('TopSpeed')
NbrKey   KEY(Names:Number)
Notes    BLOB           !Can be larger than 64K
Rec       RECORD
Name     STRING(20)
Number   SHORT
        . .

BlobSize LONG
BlobBuffer1 STRING(65520),STATIC !Maximum size string
BlobBuffer2 STRING(65520),STATIC !Maximum size string

WinView  WINDOW('View BLOB Contents'),AT(0,0,320,200),SYSTEM
        TEXT,AT(0,0,320,180),USE(BlobBuffer1),VSCROLL
        TEXT,AT(0,190,320,180),USE(BlobBuffer2),VSCROLL,HIDE
        END
CODE
OPEN(Names)
SET(Names)
NEXT(Names)
OPEN(WinView)
BlobSize = Names.Notes{PROP:Size}    !Get size of BLOB contents
IF BlobSize > 65520
    BlobBuffer1 = Names.Notes[0 : 65519]
    BlobBuffer2 = Names.Notes[65520 : BlobSize - 1]
    WinView{PROP:Height} = 400
    UNHIDE(?BlobBuffer2)
ELSE
    BlobBuffer1 = Names.Notes[0 : BlobSize - 1]
END
ACCEPT
END
```

PROP:SnapHeight and PROP:SnapWidth

PROP:SnapHeight sets the snap height of any window or dockable toolbox when it is resized. When a toolbox is repositioned from vertical docking to horizontal docking, it is resized and adjusted according to the PROP:SnapHeight and PROPSnapWidth values.

PROP:SnapWidth sets the snap width of any window or dockable toolbox when it is resized. When a toolbox is repositioned from vertical docking to horizontal docking, it is resized and adjusted according to the PROP:SnapHeight and PROP:SnapWidth values.

See Also: DOCK, DOCKED, TOOLBOX

Example:

```

PROGRAM
MAP
  MAIN
  P1 (STRING)
END

CODE
MAIN

MAIN PROCEDURE

Window APPLICATION('SnapWidth & SnapHeight'),AT(,,262,243),STATUS,MAX,RESIZE
  END
CODE
OPEN(Window)
Window{PROP:StatusText} = 'Drag the toolbar to any edge'
ACCEPT
  IF EVENT()=Event:OpenWindow
    START(p1,,system{PROP:target})
  END
END

P1 PROCEDURE(w)
win &WINDOW
width  SIGNED
height SIGNED
window WINDOW,AT(50,50,100,20),SYSTEM,TOOLBOX,GRAY,RESIZE,MDI,IMM
  BUTTON,AT(80,2,15,15),USE(?Button4),ICON(ICON:Cut),FLAT
  BUTTON,AT(31,2,15,15),USE(?Button2),ICON(ICON:Copy),FLAT
  BUTTON,AT(55,2,15,15),USE(?Button3),ICON(ICON:Paste),FLAT
END

```

```
!*****
! Using SnapWidth and SnapHeight, sets up "n" possible sizes for a TOOLBOX.
! In this example, when the toolbox is docked to the left or right, it takes
! on the Tall size when Top or Botton, it takes the Wide size.
!*****
```

```
CODE
```

```
win &= (w)
```

```
OPEN(window, win)
```

```
Window{PROP:docked} = Dock:Float
```

```
Window{PROP:dock} = DOCK:ALL
```

```
Window{PROP:snapwidth, 1} = 20 ! Vertical size i.e., when made tall
```

```
Window{PROP:snapheight, 1} = 150
```

```
Window{PROP:snapwidth, 2} = 100 ! Horizontal size i.e., when made wide
```

```
Window{PROP:snapheight, 2} = 20
```

```
Window{PROP:snapwidth, 3} = 50 ! square
```

```
Window{PROP:snapheight, 3} = 50
```

```
ACCEPT
```

```
IF EVENT() = EVENT:sized
```

```
! handle repositioning of buttons here
```

```
IF window{PROP:Width} = 20 ! is it tall?
```

```
?Button4{PROP:xpos} = 2
```

```
?Button4{PROP:ypos} = 53
```

```
?Button2{PROP:xpos} = 2
```

```
?Button2{PROP:ypos} = 19
```

```
?Button3{PROP:xpos} = 2
```

```
?Button3{PROP:ypos} = 36
```

```
ELSIF window{PROP:Width} = 100 ! Is it wide?
```

```
?Button4{PROP:xpos} = 70
```

```
?Button4{PROP:ypos} = 2
```

```
?Button2{PROP:xpos} = 50
```

```
?Button2{PROP:ypos} = 2
```

```
?Button3{PROP:xpos} = 30
```

```
?Button3{PROP:ypos} = 2
```

```
ELSE
```

```
!it must be square
```

```
?Button4{PROP:xpos} = 27
```

```
?Button4{PROP:ypos} = 27
```

```
?Button2{PROP:xpos} = 27
```

```
?Button2{PROP:ypos} = 10
```

```
?Button3{PROP:xpos} = 5
```

```
?Button3{PROP:ypos} = 27
```

```
END
```

```
END
```

```
END
```

PROP:StatusFont

A SYSTEM or WINDOW property that reads and writes the font settings for status bars. This property is used in conjunction with PROP:FontName, PROP:FontSize, PROP:FontColor, and PROP:FontStyle. The status bar uses the default character set (a value of the SYSTEM{PROP:CharSet}).

Example:

```
! Tahoma is set as a default status bar font
SYSTEM{PROP:FontName + PROP:StatusFont} = 'Tahoma'
! When this window is active, Tahoma 10 is used for status bar
Window{PROP:FontSize + PROP:StatusFont} = 10
```

PROP:StopHook

A property of the SYSTEM built-in variable that sets the override procedure for the STOP internal Clarion procedure. Equivalent to {PROP:LibHook,7}. Assign the ADDRESS of the overriding procedure, and the runtime library will call the overriding procedure instead of the STOP procedure. Assign zero and the runtime library will once again call its internal procedure. The overriding procedure's prototype must be exactly the same as the STOP procedure. (WRITE-ONLY)

PROP:SystemPropHook

A property of the SYSTEM built-in variable that allows you to trap when SYSTEM{property} is being called. Equivalent to {PROP:LibHook,12}. You need to pass the address of a procedure which has a prototype of

```
PROCEDURE(SIGNED Property, BYTE DoingSet, BYTE *Done, STRING Value), STRING
```

Property is the equated value between the property brackets {}.

DoingSet is set to TRUE (1) if SYSTEM{property} = 'value' is being called, or FALSE (0) if 'value = SYSTEM{property}' is being called.

Done is set to TRUE if you do not want the system to process the property.

Value is set to what the property is being set to. This is only set when *DoingSet* is TRUE.

The procedure must return a string. This string is only used when *DoingSet* is FALSE and *Done* is set to TRUE. In this case it is what *value* gets set to in the line

```
'value = SYSTEM{property}'.
```

PROP:TabRows

PROP:TabRows returns the number of rows of TABs in a SHEET. (READ-ONLY)

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
SHEET,AT(0,0,320,175),USE(SelectedTab)
TAB('Tab One'),USE(?TabOne)
    OPTION('Option 1'),USE(OptVar1),KEY(F10Key),HLP('Option1Help')
    RADIO('Radio 1'),AT(20,0,20,20),USE(?R1)
    RADIO('Radio 2'),AT(40,0,20,20),USE(?R2)
END
OPTION('Option 2'),USE(OptVar2),MSG('Option 2')
RADIO('Radio 3'),AT(60,0,20,20),USE(?R3)
RADIO('Radio 4'),AT(80,0,20,20),USE(?R4)
END
END
TAB('Tab Two'),USE(?TabTwo)
    PROMPT('Enter Data:'),AT(100,100,20,20),USE(?P1)
    ENTRY(@S8),AT(100,140,32,20),USE(E1)
    PROMPT('Enter More Data:'),AT(100,200,20,20),USE(?P2)
    ENTRY(@S8),AT(100,240,32,20),USE(E2)
END
TAB('Tab Three'),USE(?TabThree)
    OPTION('Option 3'),USE(OptVar3)
    RADIO('Radio 1'),AT(20,0,20,20),USE(?R5)
    RADIO('Radio 2'),AT(40,0,20,20),USE(?R6)
END
OPTION('Option 4'),USE(OptVar4)
RADIO('Radio 3'),AT(60,0,20,20),USE(?R7)
RADIO('Radio 4'),AT(80,0,20,20),USE(?R8)
END
END
TAB('Tab Four'),USE(?TabFour)
    PROMPT('Enter Data:'),AT(100,100,20,20),USE(?P3)
    ENTRY(@S8),AT(100,140,32,20),USE(E3)
    PROMPT('Enter More Data:'),AT(100,200,20,20),USE(?P4)
    ENTRY(@S8),AT(100,240,32,20),USE(E4)
END
END
BUTTON('Ok'),AT(100,180,20,20),USE(?Ok)
BUTTON('Cancel'),AT(200,180,20,20),USE(?Cancel)
END
CODE
OPEN(MDIChild)
MESSAGE('Number of TABs: ' & ?SelectedTab{PROP:NumTabs})
MESSAGE('Number of rows of TABs: ' & ?SelectedTab{PROP:TabRows})
ACCEPT;END
```

PROP:Target

A property of the SYSTEM built-in variable that is used to manipulate the current target. This property can be used to read or set the current target. It is equivalent to SETTARGET(target).

PROP:TemplImage

Property of an IMAGE control which returns the filename it creates for an image. For internal use in the Internet Connect templates, only.

PROP:TemplImagePath

SYSTEM property which sets or returns the path containing the temporary image files set by PROP:PrintMode or PROP:TemplImage. Equivalent to {PROP:TempPath,2}. For internal use in the Internet Connect templates only.

PROP:TemplImageStatus

Property of an IMAGE control which returns whether or not PROP:TemplImage created a new file or overwrote an existing file. For internal use in the Internet Connect templates, only.

PROP:TempNameFunc

Property of a REPORT which allows you to create your own names for the metafiles generated for the PREVIEW attribute by writing a callback function to supply the metafile name for each page of the report. The callback function must be a PROCEDURE which takes a single SIGNED parameter and returns a STRING.

To turn this on, you must assign the ADDRESS of your callback function to PROP:TempNameFunc. To turn it off, you must assign zero (0).

The report engine, when it is about to write a page of the report to disk, calls your procedure, passing it the page number, and uses the return value from your procedure as the name of the metafile (both on disk and in the PREVIEW attribute's QUEUE). The callback function must create the file to ensure that the name is available.

When using PROP:TempNameFunc, PROP:FlushPreview writes the metafiles to the printer but does not automatically delete them (you must clean them up yourself, whenever your program is finished using them).

Example:

```
MEMBER('MyApp')
MAP
PageNames PROCEDURE(SIGNED),STRING      !Callback function prototype
END

MyReport PROCEDURE
MyQueue QUEUE                          !Preview queue
STRING(64)
END
Report REPORT,PREVIEW(MyQueue)         !ReportDeclaration
END
CODE
OPEN(Report)
Report{PROP:TempNameFunc} = ADDRESS(PageNames) !Assign ADDRESS to property so
the
                                ! report engine calls PageNames to
                                ! get the name to use for each page
!Report processing code goes here
Report{PROP:TempNameFunc} = 0          !Assign zero to property to turn off
Report{PROP:FlushPreview} = TRUE       !Send the report to the printer
                                ! and the .WMF files are still on disk

PageNames PROCEDURE(PageNumber)        !Callback function for page names
NameVar STRING(260),STATIC
PageFile FILE,DRIVER('DOS'),NAME(NameVar),CREATE
Rec RECORD
F1 LONG
. .
CODE
NameVar = PATH() & '\PAGE' & FORMAT(PageNumber,@n04) & '.WMF'
CREATE(PageFile)
RETURN(NameVar)
```

PROP:TempPagePath

SYSTEM property which sets or returns the path containing the temporary files for the page images. Equivalent to {PROP:TempPath,1}. For internal use in the Internet Connect templates, only.

PROP:TempPath

Array SYSTEM property which sets or returns the path containing the temporary files for the page images or the path containing the temporary image files set by PROP:PrintMode. For internal use in the Internet Connect templates, only.

PROP:Thread

Returns the thread number of a window. This is not necessarily the currently executing thread, if you've used SETTARGET to set the TARGET built-in variable. (READ-ONLY)

Example:

```
WinView WINDOW('View'),AT(0,0,320,200),MDI,MAX,HVSCROLL,SYSTEM
      END
ToolboxThread BYTE
CODE
OPEN(WinView)
ToolboxThread = ToolboxWin{PROP:Thread}    !Get window thread number
ACCEPT
END
```

PROP:Threading

Property of the SYSTEM built-in variable which, when set to zero (0), disables all MDI behavior and turns the application into an SDI application.

Example:

```
PROGRAM
!Data declarations
CODE
IF SomeCondition = TRUE
  SYSTEM{PROP:Threading} = 0    !Set to SDI behavior
END
```

PROP:TipDelay and PROP:TipDisplay

PROP:TipDelay sets the time delay before tooltip display (TIP attribute) for the SYSTEM (16-bit only – now obsolete).

PROP:TipDisplay sets the duration of tooltip display (TIP attribute) for the SYSTEM (16-bit only – now obsolete).

Example:

```
WinView  APPLICATION('MyApp'),AT(0,0,320,200),MAX,HVSCROLL,SYSTEM
        END
```

```
CODE
OPEN(WinView)
SYSTEM{PROP:TipDelay} = 50    !Delay TIP display for 1/2 second
SYSTEM{PROP:TipDisplay} = 500 !TIP display for 5 seconds
ACCEPT
END
```

PROP:TipsFont

A property of the SYSTEM built-in variable that reads and writes the font settings for tooltips. This property is used in conjunction with PROP:FontName, PROP:FontSize, PROP:FontColor, and PROP:FontStyle. Tooltips use the default character set (a value of the SYSTEM{PROP:CharSet}).

Example:

```
SYSTEM {PROP:FontName + PROP:TipsFont} !set/get font's typeface
SYSTEM {PROP:FontSize + PROP:TipsFont} !set/get font's size
SYSTEM {PROP:FontColor + PROP:TipsFont} !set/get font's color
SYSTEM {PROP:FontStyle + PROP:TipsFont} !set/get font's style
```

PROP:Touched

When non-zero, indicates the data in the ENTRY, TEXT, SPIN, or COMBO control with input focus has been changed by the user since the last EVENT:Accepted. This is automatically reset to zero each time the control generates an EVENT:Accepted. Setting this property (in EVENT:Selected) allows you to ensure that EVENT:Accepted generates to force data validation code to execute, overriding Windows' standard behavior--simply pressing TAB to navigate to another control does not automatically generate EVENT:Accepted.

PROP:Touched can also be interrogated to determine if the content of a BLOB has changed since it was retrieved from disk.

Example:

```
WinView  WINDOW('View'),AT(0,0,320,200),MDI,MAX,HVSCROLL
          ENTRY(@S30),AT(0,0,320,180),USE(Fil:Field)
          BUTTON('Ok'),AT(100,180,20,20),USE(?Ok)
          BUTTON('Cancel'),AT(200,180,20,20),USE(?Cancel)
          END
SaveCancelPos LONG,DIM(4)
CODE
OPEN(WinView)
SaveCancelPos[1] = ?Cancel{PROP:Xpos}          !Save Cancel button area
SaveCancelPos[2] = ?Cancel{PROP:Xpos}+?Cancel{PROP:Width}
SaveCancelPos[3] = ?Cancel{PROP:Ypos}
SaveCancelPos[4] = ?Cancel{PROP:Ypos}+?Cancel{PROP:Height}
ACCEPT
CASE FIELD()
OF ?Fil:Field
CASE EVENT()
OF EVENT:Selected
  ?Fil:Field{PROP:Touched} = 1          !Force EVENT:Accepted to generate
OF EVENT:Accepted
  IF KEYCODE() = MouseLeft AND |          !Detect user clicking on Cancel
    INRANGE(MOUSEX(),SaveCancelPos[1],SaveCancelPos[2]) AND |
    INRANGE(MOUSEY(),SaveCancelPos[3],SaveCancelPos[4])
  CYCLE          !User clicked on Cancel
ELSE
!Process the data, whether entered by the user or in the field at the start
END
```

```
OF ?Ok
  CASE EVENT()
  OF EVENT:Accepted
    !Write the data to disk
  END
OF ?Cancel
  CASE EVENT()
  OF EVENT:Accepted
    !Do not write the data to disk
  END
END
END
```

PROP:Type

Contains the type of control. Values are the CREATE:xxxx equates (listed in EQUATES.CLW). (READ-ONLY)

Example:

```
MyField  STRING(1)
?MyField EQUATE(100)
```

```
WinView  WINDOW('View'),AT(0,0,320,200),MDI,MAX,HVSCROLL
        END
CODE
OPEN(WinView)
IF UserChoice = 'CheckField'
    CREATE(?MyField,CREATE:Check)
ELSE
    CREATE(?MyField,CREATE:Entry)
END
?MyField{PROP:Use} = MyField
SETPOSITION(?MyField,10,10)
IF ?MyField{PROP:Type} = CREATE:Check      !Check control type
    ?MyField{PROP:TrueValue} = 'T'
    ?MyField{PROP:FalseValue} = 'F'
END
ACCEPT
END
```

PROP:UpsideDown

This toggles both the UP and DOWN attributes at once to display inverted TAB control text in a SHEET structure.

Example:

```
WinView WINDOW('View'),AT(0,0,320,200),MDI,MAX,HVSCROLL
SHEET,AT(0,0,320,175),USE(SelectedTab),RIGHT,DOWN !Tabs right reading
down
  TAB('Tab One'),USE(?TabOne)
    PROMPT('Enter Data:'),AT(100,100,20,20),USE(?P1)
    ENTRY(@S8),AT(100,140,32,20),USE(E1)
    PROMPT('Enter More Data:'),AT(100,200,20,20),USE(?P2)
    ENTRY(@S8),AT(100,240,32,20),USE(E2)
  END
  PROMPT('Enter Data:'),AT(100,100,20,20),USE(?P3)
  ENTRY(@S8),AT(100,140,32,20),USE(E3)
  PROMPT('Enter More Data:'),AT(100,200,20,20),USE(?P4)
  ENTRY(@S8),AT(100,240,32,20),USE(E4)
  END
  END
  BUTTON('Ok'),AT(100,180,20,20),USE(?Ok)
  BUTTON('Cancel'),AT(200,180,20,20),USE(?Cancel)
  END
CODE
OPEN(WinView)
?SelectedTab{PROP:BELOW} = TRUE !Set tabs to display at bottom of sheet
?SelectedTab{PROP:UpsideDown} = TRUE !Invert the text displayed on the tabs
ACCEPT
END
```

PROP:UseAddress

Returns the address of an ANY typed variable that refers to the USE variable of a control.

Example:

```
Loc:Any ANY
Window WINDOW('Caption'),AT(,,260,100),GRAY
  ENTRY(@s20),AT(57,23,60,10),USE(Loc:Any)
END
CODE
OPEN(Window)
ACCEPT
  X# = ?Loc:Any{PROP:UseAddress}
  BREAK
END
```

PROP:Visible

Returns an empty string if the control is not visible because either because it has been hidden, or it is a member of a "parent" control (OPTION, GROUP, MENU, SHEET, or TAB) that is hidden, or is on a TAB control page that is not currently selected. (READ-ONLY)

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
SHEET,AT(0,0,320,175),USE(SelectedTab)
TAB('Tab One'),USE(?TabOne)
    PROMPT('Enter Data:'),AT(100,100,20,20),USE(?P1)
    ENTRY(@S8),AT(100,140,32,20),USE(E1)
    PROMPT('Enter More Data:'),AT(100,200,20,20),USE(?P2)
    ENTRY(@S8),AT(100,240,32,20),USE(E2)
    END
TAB('Tab Two'),USE(?TabTwo)
    PROMPT('Enter Data:'),AT(100,100,20,20),USE(?P3)
    ENTRY(@S8),AT(100,140,32,20),USE(E3)
    PROMPT('Enter More Data:'),AT(100,200,20,20),USE(?P4)
    ENTRY(@S8),AT(100,240,32,20),USE(E4)
    END
END
END
BUTTON('Ok'),AT(100,180,20,20),USE(?Ok)
BUTTON('Cancel'),AT(200,180,20,20),USE(?Cancel)
END
CODE
OPEN(MDIChild)
ACCEPT
CASE EVENT()
OF EVENT:Completed
    BREAK
END
CASE FIELD()
OF ?Ok
    CASE EVENT()
    OF EVENT:Accepted
        SELECT
        END
    OF ?E3
        CASE EVENT()
        OF EVENT:Accepted
            E3 = UPPER(E3)          !Convert the data entered to Upper case
            IF ?E3{PROP:Visible} AND MDIChild{PROP:AcceptAll}
                !Check for visibility during AcceptAll mode
                DISPLAY(?E3)      ! and display the upper cased data
            END
        END
    END
END
```

```
OF ?Cancel
CASE EVENT( )
OF EVENT:Accepted
  BREAK
END
END
END
```

PROP:VLBproc and PROP:VLBval

PROP:VLBProc sets the source procedure for a "Virtual List Box" LIST or COMBO control without a FROM attribute. This procedure provides the control with the data to display.

The procedure's prototype must take three parameters:

```
VLBProc PROCEDURE(LONG, LONG, SHORT), STRING
```

where the first LONG is either SELF (indicating the procedure is a method of a CLASS) or the value set for PROP:VLBval. The second LONG passes the row number of the virtual list box to affect. There are three "special" values for this parameter, -1 asks for the number of records to display in the list, -2 asks for the number of fields in the nominal Queue (data and color/tree/icon fields) to display in the list, and -3 asks if there are any changes to display. The SHORT parameter specifies the column number of the virtual list box to affect.

PROP:VLBVal sets the source object for a "Virtual List Box" LIST or COMBO control without a FROM attribute. This can be any 32-bit unique value to identify the specific list box, but is generally the return value of ADDRESS(SELF) when the PROP:VLBProc procedure is a CLASS method.

Example:

```
PROGRAM
MAP
  Main
END

StripedListQ  QUEUE,TYPE
S              STRING(20)
              END

StripedList   CLASS,TYPE
Init          PROCEDURE(WINDOW w, SIGNED feq, StripedListQ Q)
VLBproc       PROCEDURE(LONG row, SHORT column),STRING,PRIVATE
              !Required first parameter is implicit in a CLASS method
Q             &StripedListQ,PRIVATE
ochanges      LONG,PRIVATE
              END

CODE
  Main
```

```

StripedList.Init PROCEDURE(WINDOW w, SIGNED feq, StripedListQ Q)

CODE
SELF.Q &= Q
SELF.ochanges = CHANGES(Q)

w $ feq{PROP:VLBval} = ADDRESS(SELF)           !Must assign this first
w $ feq{PROP:VLBproc} = ADDRESS(SELF.VLBproc) ! then this

StripedList.VLBproc PROCEDURE(LONG row, SHORT col) !Required first parameter is impleie
nchanges LONG

CODE
CASE row
OF -1           ! How many rows?
RETURN RECORDS(SELF.Q)
OF -2           ! How many columns?
RETURN 5        ! 1 data, four color fields in the "nominal Q"
OF -3           ! Has it changed
nchanges = CHANGES(SELF.Q)
IF nchanges <> SELF.ochanges THEN
SELF.ochanges = nchanges
RETURN 1
ELSE
RETURN 0
END
ELSE
GET(SELF.Q, row)
CASE col
OF 1           !Data field
RETURN WHAT(SELF.Q,1)
OF 3           !Background color field
RETURN CHOOSE(BAND(row,1), COLOR:none, 0c00000H)
ELSE           !All other fields
RETURN COLOR:None           ! Use default color
END
END

```

```
Main PROCEDURE
window WINDOW('Caption'),AT(,,153,103),GRAY,SYSTEM
      LIST,AT(33,12,80,80),USE(?List1),FORMAT('20L*')
      END
Q      QUEUE(StripedListQ)
      END
SL     StripedList
i      SIGNED
CODE
LOOP i = 1 TO 20
  Q.s = 'Line ' & i
  ADD(Q)
END
OPEN(window)
SL.Init(window, ?list1, Q)
ACCEPT
END
```

PROP:Vscroll

This Read/Write property mimics the VSCROLL attribute. If the value is set to zero (0), the vertical scrollbar will not be displayed, even if the number of records in the list is more than the number of visible rows.

When you place VSCROLL on a LIST with the IMM attribute, the vertical scroll bar is always present, even when the list is not full. When the user clicks on the scroll bar, events are generated, but the list contents do not move (executable code should perform this task). You can interrogate the PROP:VscrollPos property to determine the scroll thumb's position in the range 0 (top) to 100 (bottom).

PROP:VscrollPos

Returns the position of the vertical scroll bar's "thumb." Valid values are from 0 to 255 on a window, IMAGE, or TEXT control with the VSCROLL attribute.

Valid values are from 0 to 100 on a LIST, or COMBO control with the VSCROLL attribute (when the IMM attribute is set on). If the IMM attribute is omitted, the values range from 0 to the number of items available for display in the LIST or COMBO (as specified by the FROM() attribute) minus one. If FROM(Queue) is specified, then the range is 0 to RECORDS(Queue) - 1. If FROM('list'), then the range is 0 to the number of items in 'list' - 1. For example, if FROM('Mr.|Mrs.|Ms.|Dr. '), Then the range is 0 to 3, because 4 is the number of items in the list.

The value of PROP:VScrollPos with IMM off specifies the exact number of items in the list or QUEUE that *precede* the first item displayed. For example, if row 5 of a QUEUE is the first item displayed, then the value of PROP:VScrollPos is 4.

Setting this property causes the control or window's contents to be scrolled vertically (unless the IMM attribute is on the LIST or COMBO, then only the "thumb" moves).

Example:

```

Que  QUEUE
      STRING(50)
      END
WinView WINDOW('View'), AT(0,0,320,200), MDI, SYSTEM
      LIST, AT(0,0,320,200), USE(?List), FROM(Que), IMM, VSCROLL
      END

CODE
OPEN(WinView)
Fil:KeyField = 'A' ; DO BuildListQue
ACCEPT
CASE FIELD()
OF ?List
CASE EVENT()
OF EVENT:ScrollDrag
EXECUTE INT(?List{PROP:VscrollPos}/10) + 1
  Fil:KeyField = 'A'
  Fil:KeyField = 'C'
  Fil:KeyField = 'E'
  Fil:KeyField = 'G'
  Fil:KeyField = 'K'
  Fil:KeyField = 'M'
  Fil:KeyField = 'P'
  Fil:KeyField = 'S'
  Fil:KeyField = 'V'
  Fil:KeyField = 'Y'
END
DO BuildListQue
END ; END; END

```

```
FREE(Que)

BuildListQue ROUTINE
FREE(Queue)
SET(Fil:SomeKey,Fil:SomeKey)           !Set to selected key field
LOOP ?List{PROP:Items} TIMES          !Process number of recs visible
in list
  NEXT(SomeFile) ; IF ERRORCODE() THEN BREAK. !Break at end of file
  Que = Fil:KeyField                   !Assign field to display to
QUEUE
  ADD(Que)                             ! and add it to the QUEUE
END
```

See Also: X and Y Origin

PROP:Watched

Property of a FILE that returns whether the current record is being WATCHed. Returns 1 if the record is watched and an empty string (") if not. (READ ONLY)

Example:

```
FileName STRING(256)
Customer FILE,DRIVER('Clarion')
Record RECORD
Name STRING(20)
. .
CODE
OPEN(Customer)
SET(Customer)
LOOP
WATCH(Customer)
NEXT(Customer)
IF ERRORCODE() THEN BREAK.
IF Customer{PROP:Watched} <> ''
MESSAGE('Record watched')
END
END
```

PROP:WheelScroll

A read/write property valid for list and combo box controls. Used to adjust the control sensitivity to mouse wheel scrolling. If property is set to a value less than 1, the property defaults to 1. The lower the value of the property results in more lines scrolled per wheel "click". Default value is 40 (approximately 3 rows per wheel "click" for standard mouse driver settings).

PROP:WindowsVersion

Returns the string that describes Windows version running the program.

Read only. Available for SYSTEM only.

Example:

```
GLO:WindowsVersion = SYSTEM{PROP:WindowsVersion}
```

PROP:WndProc

Sets or gets the window's (not the client area) or a specific control's messaging procedure for use with low-level Windows API calls that require it. Generally used in sub-classing to track all Windows messages.

Example:

```

PROGRAM
MAP
Main      PROCEDURE
SubClassFunc1  PROCEDURE(USHORT,SHORT,USHORT,LONG),LONG,PASCAL
SubClassFunc2  PROCEDURE(USHORT,SHORT,USHORT,LONG),LONG,PASCAL
      MODULE('Windows')                                !SoftVelocity Win31
Library
CallWindowProc PROCEDURE(LONG,UNSIGNED,SIGNED,UNSIGNED,LONG),LONG,PASCAL
. . .
      !End MAP and MODULE
SavedProc1     LONG
SavedProc2     LONG
WM_MOUSEMOVE   EQUATE(0200H)
PT             GROUP
X              SHORT
Y              SHORT
      END
CODE
Main
Main          PROCEDURE
WinView WINDOW('View'),AT(0,0,320,200),HVSCROLL,MAX,TIMER(1),STATUS
      STRING('X Pos'),AT(1,1,,),USE(?String1)
      STRING(@n3),AT(24,1,,),USE(PT:X)
      STRING('Y Pos'),AT(44,1,,),USE(?String2)
      STRING(@n3),AT(68,1,,),USE(PT:Y)
      BUTTON('Close'),AT(240,180,60,20),USE(?Close)
      END
CODE
OPEN(WinView)
SavedProc1 = WinView{PROP:WndProc}          !Save this procedure
WinView{PROP:WndProc} = ADDRESS(SubClassFunc1) !Name subclass procedure
SavedProc2 = WinView{PROP:ClientWndProc}    !Save this procedure
WinView{PROP:ClientWndProc} = ADDRESS(SubClassFunc2) !Name subclass procedure
ACCEPT
CASE ACCEPTED()
OF ?Close
  BREAK
END
END

```

```
SubClassFunc1    PROCEDURE(hWnd,wMsg,wParam,lParam)    !Sub class procedure
CODE            ! to track mouse movement in    IF wMsg =
WM_MOUSEMOVE    ! window's status bar (only)
    PT.X = MOUSEX() ; PT.Y = MOUSEY()            !Assign mouse position
END
RETURN(CallWindowProc(SavedProc1,hWnd,wMsg,wParam,lParam))!Pass control to
SavedProc1
```

```
SubClassFunc2    PROCEDURE(hWnd,wMsg,wParam,lParam)    !Sub class procedure
CODE            ! to track mouse movement in    IF wMsg =
WM_MOUSEMOVE    ! window's client area
    PT.X = MOUSEX() ; PT.Y = MOUSEY()            !Assign mouse position
END
RETURN(CallWindowProc(SavedProc2,hWnd,wMsg,wParam,lParam))!Pass control to
SavedProc2
```

PROP:Xorigin and PROP:Yorigin

Attributes of a SHEET or IMAGE control (which have horizontal and/or vertical scrollbar(s)) that sets or retrieves how much the SHEET or IMAGE has been scrolled. XORIGIN (PROP:Xorigin) specifies how much the SHEET or IMAGE has scrolled in the X direction. YORIGIN (PROP:Yorigin) specifies how much the SHEET or IMAGE has scrolled in the Y direction. As the example shows below, these properties are also valid on a WINDOW.

The Read-only property **PROP:Yorigin** also can return the value that CHOICE() would return for the first visible row of a referenced list box. This property allows to calculate the current position of the highlighted row, (e.g. CHOICE(?List) - ?List{PROP:Yorigin} + 1)

Example

```

PROGRAM
MAP
END
S STRING(20)
Window WINDOW('Caption'),AT(,,400,250),FONT('MS Sans Serif',8,,FONT:regular)|
      ,IMM,VSCROLL, STATUS,SYSTEM,GRAY,CENTER
      ENTRY(@S20),AT(10,100),USE(S,,?S1),SCROLL
      ENTRY(@S20),AT(10,500),USE(S,,?S2),SCROLL
      ENTRY(@S20),AT(10,900),USE(S,,?S3),SCROLL
END

y1      UNSIGNED
y2      UNSIGNED
Wy1     UNSIGNED
Wy2     UNSIGNED
Control SIGNED

CODE
OPEN(Window)
ACCEPT
CASE EVENT()
OF EVENT:Selected
  Control = SELECTED()
  IF Control
    y1 = Control {PROP:yPos}
    y2 = y1 + Control {PROP:Height}
    Wy1 = 0{PROP:Yorigin}           !read Yorigin
    Wy2 = Wy1 + 0 {PROP:ClientHeight}
    IF y1 < Wy1
      0{PROP:Yorigin} = CHOOSE (y1 < 20, 0, y1 - 20)
    ELSIF y2 > Wy2
      0{PROP:Yorigin} = Wy1 + y2 - Wy2 + 20       !set Yorigin
    END
  END
END
END
END
CLOSE(Window)

```

Runtime VIEW and FILE Properties

PROP:Completed

PROP:Completed is a property of a FILE that returns the percentage completed of the re-build during a BUILD or PACK operation for which PROP:ProgressEvents has been turned on. Returns zero (0) if the file driver does not know how much of the BUILD or PACK has been done. (READ ONLY)

PROP:ConnectionString

Property of a FILE using the ODBC driver that returns the connection string (normally stored in the file's OWNER attribute) that would allow a complete connection. If the OWNER attribute contains only a data source name, a login screen appears to ask for the rest of the required details before the connection is made. This login window appears every time you log on. With this property, the developer can enter information in the login screen once, then set the OWNER attribute to the return value from PROP:ConnectionString, eliminating the login.

Example:

```
OwnerString STRING(20)
Customer FILE,DRIVER('ODBC'),OWNER(OwnerString)
Record RECORD
Name STRING(20)
. .
CODE
OwnerString = 'DataSourceName'
OPEN(Customer)
OwnerString = Customer{PROP:ConnectionString} !Get full connect string
MESSAGE(OwnerString) !Display it for future use
```

PROP:CurrentKey

Property of a FILE that returns a reference to the current KEY or INDEX being used for sequential processing, or the current key being built during a BUILD or PACK operation (READ ONLY). Valid only as the source side of a reference assignment statement or in a logical expression comparing the return result to NULL. Returns NULL if the file is being processed in record order.

Example:

```
KeyRef &KEY
Customer FILE,DRIVER('Clarion'), PRE(Cus)
NameKey KEY(Cus:Name),DUP
Record RECORD
Name STRING(20)
. .
CODE
OPEN(Customer)
SET(Customer)
KeyRef &= Customer{PROP:CurrentKey} !Returns NULL
IF Customer{PROP:CurrentKey} &= NULL !Compare to NULL
MESSAGE('SET to record order')
END
SET(Cus:NameKey)
KeyRef &= Customer{PROP:CurrentKey} !Returns reference to Cus:NameKey
```

PROP:DataPath

A read/write SYSTEM property that can be used to set the default directory for data files.

All files with unqualified file names (e.g., those files with no NAME attribute or a NAME with only a relative path) will be looked for in the directory specified by SYSTEM{PROP:DataPath}.

SYSTEM{PROP:DataPath} defaults to the directory the application starts in. This will save developers from having to do startup code like the following:

```
GETINI(datadirectory)
filename = datadirectory & 'file1'
file2name = datadirectory & 'file2'
file3name = datadirectory & 'file3'
```

PROP:Details

Property of a FILE that toggles turing Record Buffer logging on or off. file{PROP:Details}=1 turns Record Buffer logging on and file{PROP:Details}=0 turns Record Buffer logging off.

See Also:

PROP:Profile

PROP:Log

PROP:DriverLogoutAlias

Property of a FILE that returns whether the file driver allows the LOGOUT statement to name both a file and an alias for the file in the same statement (READ ONLY).

Example:

```
IF Customer{PROP:DriverLogoutAlias} = '' !Test for alias allowed in LOGOUT
  MESSAGE('Driver does not allow files and their aliases in LOGOUT')
END
```

PROP:FetchSize

Property of a FILE or VIEW which sets or gets the *pagesize* parameter for the last BUFFER statement executed.

Example:

```
CODE
OPEN(MyFile)
BUFFER(MyFile,10,5,2,300) !10 records per page, 5 pages behind and 2 read-
ahead,
                ! with a 5 minute timeout
MyFile{PROP:FetchSize} = 1 !Change fetch rate to one record at a time
```

PROP:Field

Property of a VIEW which returns the field number in a file for the specified field number in a VIEW. (READ ONLY)

PROP:FieldsFile

Property of a VIEW which returns a reference to the file that contains the specified field number in a VIEW. (READ ONLY)

Example:

```
Header  FILE,DRIVER('TopSpeed'),PRE(Hea)
OrderKey KEY(Hea:AcctNumber, Hea:OrderNumber)
Record  RECORD
AcctNumber  LONG
OrderNumber  LONG
OrderDate   LONG
. .

Detail  FILE,DRIVER('TopSpeed'),PRE(Det)
OrderKey KEY(Det:AcctNumber,Det:OrderNumber)
Record  RECORD
AcctNumber  LONG
OrderNumber  LONG
ItemNumber  LONG
. .

AView  VIEW(Header)
        PROJECT(Hea:AcctNumber,Hea:OrderNumber,Hea:OrderDate)
        JOIN(Det:OrderKey,Hea:AcctNumber,Hea:OrderNumber)
. .

!AView{PROP:Field,1} Returns 1 which represents Hea:AcctNumber
!AView{PROP:Field,2} Returns 2 which represents Hea:OrderNumber
!AView{PROP:Field,3} Returns 3 which represents Hea:OrderDate
!AView{PROP:Field,4} Returns 1 which represents Det:AcctNumber
!AView{PROP:Field,5} Returns 2 which represents Det:OrderNumber
!AView{PROP:Field,6} Returns 3 which represents Det:ItemNumber

!AView{PROP:FieldsFile,2} Returns a reference to the Header File
!AView{PROP:FieldsFile,5} Returns a reference to the Detail File
```

PROP:File

An array property of a VIEW. Each array element returns a reference to the numbered file in the VIEW. This reference can be used as the source side of a reference assignment statement. The files are numbered within the VIEW starting with 1 (the primary file in the VIEW) and continuing for each JOIN, as they appear within the VIEW structure. (READ ONLY)

PROP:File can now also be used to return the key's file name from a KEY.

Example:

```
GetFileInfoFromKey PROCEDURE(*KEY key)
KFile &FILE
Record &GROUP
CODE
KFile &= key{PROP:File}
MESSAGE(KFile{PROP:Label, key{PROP:Field, 1}}) !Returns label of first key component
```

PROP:Files

Property of a VIEW which returns the total number of files in the VIEW. This is equivalent to the total number of JOIN structures, plus one (the primary file named in the VIEW statement itself). (READ ONLY)

Example:

```
AView VIEW(BaseFile) !File 1
JOIN(ParentFile,'BaseFile.parentID = ParentFile.ID') !File 2
JOIN(GrandParent.PrimaryKey, ParentFile.GrandParentID) !File 3
END
END
JOIN(OtherParent.PrimaryKey,BaseFile.OtherParentID) !File 4
END
END ! AView{PROP:Files} returns 4
! AView{PROP:File,1} returns a reference to BaseFile
! AView{PROP:File,2} returns a reference to Parent
! AView{PROP:File,3} returns a reference to GrandParent
! AView{PROP:File,4} returns a reference to OtherParent

FilesQ QUEUE
FileRef &FILE
END
CODE
LOOP X# = 1 TO AView{PROP:Files} !Loop 4 times
FilesQ.FileRef &= AView{PROP:File,X#} !Reference assign each file in the VIEW
ADD(FilesQ) ! and add it to the queue
ASSERT(~ERRORCODE()) !Assume no errors
CLEAR(FilesQ) !Clear the queue for the next assignment
END
```

PROP:GlobalHelp

SYSTEM property which, when on, specifies disabling automatic .HLP file closing when the window which opened the .HLP file is closed. This makes the .HLP file stay open until the user closes it.

Example:

```
SYSTEM{PROP:GlobalHelp} = TRUE      !Disable automatic HLP file close
```

PROP:Held

Property of a FILE that returns whether the current record is held. Returns 1 if the record is held and an empty string ("") if not. (READ ONLY)

Example:

```
FileName STRING(256)
Customer FILE,DRIVER('Clarion')
Record RECORD
Name STRING(20)
. .
CODE
OPEN(Customer)
SET(Customer)
LOOP
HOLD(Customer,1)
NEXT(Customer)
IF ERRORCODE() THEN BREAK.
IF Customer{PROP:Held} <> ''
MESSAGE('Record Held')
END
END
```

PROP:Log

Property of a FILE that writes a string to the current profile output file (assigned to PROP:Profile). This string is placed on its own line in the file. (WRITE ONLY)

Example:

```
FileName STRING(256)
Customer FILE,DRIVER('TopSpeed')
Record RECORD
Name STRING(20)
    END
    END

CODE
Customer{PROP:Profile} = 'CustLog.TXT'
!statement above turns profiling on, output file:CustLog.TXT
OPEN(Customer)
Filename = Customer{PROP:Profile} !Get name of current log file

Customer{PROP:Log} = CLIP(FileName) & ' ' & |
    FORMAT(TODAY(),@D2) & ' ' & |
    FORMAT(CLOCK(),@T1)
!Write a line of text to the log file

SET(Customer)
LOOP
    NEXT(Customer) !All file I/O action is logged out to
    IF ERRORCODE() THEN BREAK. ! the CustLog.TXT file
END
Customer{PROP:Profile} = '' !Turn profiling off
```

PROP:Logout

Property of a FILE that assigns or returns the priority level of the referenced FILE within a transaction. out may be used to build the list of files in the transaction before issuing the LOGOUT(*seconds*) statement to begin the transaction. By using PROP:Logout, you can add more files to the transaction than the limited number of parameters the LOGOUT statement will allow. If the LOGOUT statement lists any files at all, all files previously set for the transaction by PROP:Logout are removed from the transaction and only the files listed in the LOGOUT statement are logged out.

The priority level indicates the order in which the file is logged out in the transaction, with lower numbers being logged out before the higher numbers. If two files have the same priority level, they are logged out in the order in which they were added to the logout list. Assigning a positive priority level adds the FILE to the transaction, assigning a negative priority level removes the FILE from the transaction, and assigning zero (0) has no effect. Querying PROP:Logout returns the priority level assigned to the file, and zero (0) if the file is not a part of the transaction.

Attempting to use PROP:Logout to add a file to the transaction which uses a different file driver will result in ERRORCODE 48, "Unable to log transaction."

Example:

```
Customer FILE,DRIVER('TopSpeed')
Record RECORD
CustNumber LONG
Name STRING(20)
. .
Orders FILE,DRIVER('TopSpeed')
Record RECORD
CustNumber LONG
OrderNumber LONG
OrderDate LONG
. .
Items FILE,DRIVER('TopSpeed')
Record RECORD
OrderNumber LONG
ItemNumber LONG
. .
CODE
Customer{PROP:Logout} = 1 !Add Customer file to logout list and set priority to 1
Items{PROP:Logout} = 2 !Add Items file to logout list and set priority to 2
Orders{PROP:Logout} = 1 !Add Orders file to logout list and set priority to 1
X# = Items{PROP:Logout} !Return Items file priority level (X# = 2)
Customer{PROP:Logout} = -1 !Remove Customer file from logout list
LOGOUT(1) !Begin transaction and
! logout files in this order: Orders, Items
COMMIT !Terminate the transaction
```

PROP:LogoutIsolationLevel

Property of a FILE that controls what isolation level is used within a transaction frame. This property is valid on all SQL based drivers (except Oracle). The default value is 8.

PROP:LogoutIsolationLevel uses the ODBC isolation level standard. This may not be the same as the isolation levels documented on the target driver's native back end. For example, with Sybase's ASA, the documented isolation levels are 0, 1, 2 and 3 and they correspond to

ODBC level	Sybase Level
1	0
2	1
4	2
8	3
16	N/A

PROP:MaxStatements

Property of a FILE that allows you to find out how many statements can be active before a new connection to the database is raised. This property can also set the maximum number of statements before another connection is raised. This property is valid on all SQL file types except Oracle.

PROP:Profile

Property of a FILE that toggles logging out (profiling) all file I/O calls and errors returned by the file driver to a specified text file. Assigning a filename to PROP:Profile initiates profiling, while assigning an empty string ("") turns off profiling. Querying this property returns the name of the current logfile, and an empty string ("") if profiling is turned off.

PROP:ProgressEvents

PROP:ProgressEvents is a property of a FILE that generates events to the currently open window during a BUILD or PACK operation (WRITE ONLY). This property is driver-dependent, see the file driver's documentation for support.

Assigning a value of zero (0) turns off event generation for the next BUILD or PACK statement executed, while assigning any other value (valid range--1 to 100) turns on event generation. Out of range assignments are treated as follows: a negative number is treated as one (1), and any value greater than one hundred (100) is treated as one hundred (100). The larger the value assigned, the more events are generated and the slower the BUILD or PACK will progress.

Events generated are: EVENT:BuildFile, EVENT:BuildKey, and EVENT:BuildDone. It is not valid to make any calls to the FILE being built except to query its properties, call NAME(*file*), or CLOSE(*file*) (which aborts the process and is not recommended). Issuing a CYCLE statement in response to any of the events generated (except EVENT:BuildDone) cancels the operation.

PROP:CurrentKey may be used to get a reference to the current key being built, then PROP:Label may be used to retrieve the key's label for display to the user.

PROP:Completed is a property of a FILE that returns the percentage completed of the re-build during a BUILD or PACK operation for which PROP:ProgressEvents has been turned on. Returns zero (0) if the file driver does not know how much of the BUILD or PACK has been done. (READ ONLY)

Example:

```

PROGRAM
MAP.
INCLUDE('ERRORS.CLW')
Test FILE,DRIVER('TOPSPEED','/FULLBUILD=ON'),CREATE,PRE(TEST)
K1      KEY(Test:Xval)
        RECORD
Xval    LONG
. .
counter    LONG
CurrentKey &KEY
cancelling BYTE(FALSE)
BuildDone  BYTE(FALSE)
Completed  LONG(1)
CurEvent  LONG
window WINDOW('Time Slicing Build Example'),AT(,,127,68),SYSTEM,GRAY
        STRING('Building'),AT(9,6),USE(?BuildStr)
        STRING(''),AT(39,6),USE(?Name)
        PROGRESS,USE(counter),AT(9,25,107,8),RANGE(0,100)
        BUTTON('&Cancel'),AT(82,45),USE(?Cancel),DISABLE
END

```

```

CODE
OPEN(Test)
IF ERRORCODE()
  CREATE(Test); OPEN(Test); STREAM(Test)
  LOOP 20000 TIMES
    Test.Xval = X#; X# += 1; APPEND(Test)
  END
  FLUSH(Test)
END
OPEN(window)
ACCEPT
CurEvent = EVENT()
CASE CurEvent
  OF EVENT:OpenWindow
    Test{PROP:ProgressEvents} = 100           !Turn on event generation
    BUILD(Test)
    ENABLE(?Cancel)
  OF EVENT:Accepted
    IF ACCEPTED() = ?Cancel
      IF BuildDone THEN BREAK.
      IF MESSAGE('Cancelling build leaves file unusable. Cancel anyway?'|
        , 'Warning', ICON:Exclamation, BUTTON:Yes+BUTTON:No, BUTTON:No) |
        = BUTTON:Yes
        Cancelling = TRUE
        ?BuildStr{PROP:Text} = 'Please Wait. Cancelling Build'
        ?Name{PROP:Text} = ''
        DISPLAY(?BuildStr, ?Name)
      END
    END
  OF EVENT:BuildFile
  OROF EVENT:BuildKey           !Process BUILD events
    IF Cancelling = TRUE; DO Done; CYCLE.
    IF CurEvent = EVENT:BuildKey
      CurrentKey &= Test{PROP:CurrentKey}      !Get current key reference
      IF NOT (CurrentKey &= NULL)
        ?Name{PROP:Text} = CurrentKey{PROP:Label} !Display key name
      END
    ELSE
      ?Name{PROP:Text} = NAME(Test)
    END
    IF Completed<>0; Completed=Test{PROP:Completed}.!Get completion percentage
    IF Completed = 0
      counter += 10
      IF (counter>100) THEN counter = 0.
    ELSE
      counter = Completed
    END
  END
  DISPLAY(?Name, ?Counter)

```

```
    OF EVENT:BuildDone
      DO Done
      END
    END
  OPEN(Test)
  IF ERRORCODE() = BadKeyErr THEN MESSAGE(NAME(Test) & ' BUILD failed' ).

Done ROUTINE
  BuildDone = TRUE
  ?Cancel{PROP:Text} = '&OK'
  CLOSE(Test)
```

PROP:Record

Property of a FILE that returns a group reference to the file's record buffer. (READ ONLY)

Example:

```
Rec &GROUP
Customer FILE,DRIVER('TopSpeed')
Record RECORD
Name STRING(20)
. .
CODE
Rec &= Customer{PROP:Record}
```

PROP:SQLDriver

A of a FILE that returns '1' if the file driver accepts SQL, otherwise it returns an empty string ("). (READ ONLY)

Example:

```
Customer FILE,DRIVER('Clarion'),PRE(CUS)
Record RECORD
Name STRING(20)
. .
SQLFlag BYTE
CODE
IF Customer{PROP:SQLDriver} THEN SQLFlag = TRUE.
```

PROP:StmtAttr

Property of a FILE that allows you to read or set the ODBC Statement Attribute. This is equivalent to calling the ODBC API functions *SQLSetStmtOption* or *SQLGetStmtOption*. This property is valid on all SQL file types except Oracle.

Example:

```
Customer{PROP:StmtAttr,7} = 3 ! Sets concurrency to row version checking
```

PROP:SupportsOp

Property of a FILE that returns whether the specified function is supported by the file driver. Returns 1 if the operation is supported and 0 if it is not. Valid functions supported by this property are the DriverOp:xxxx found in EQUATES.CLW. (READ ONLY)

PROP:SupportsType

Property of a FILE that returns whether the specified data type is supported by the file driver. Returns 1 if the data type is supported and 0 if it is not. Valid data types supported by this property are the DataType:xxxx found in EQUATES.CLW. (READ ONLY)

Example:

```
Customer FILE,DRIVER('Clarion')
Record    RECORD
Id        LONG
        . .
Supported BYTE
        CODE
Supported = Customer{PROP:SupportsType,DriverOp:CREATE} ! Returns 1
Supported = Customer{PROP:SupportsType,DriverOp:NULL}   ! Returns 0
Supported = Customer{PROP:SupportsType,DataType:LONG}   ! Returns 1
Supported = Customer{PROP:SupportsType,DataType:DATE}   ! Returns 0
```

PROP:Text (MEMO Property)

An array property of a FILE that sets or returns the specified MEMO field's data. MEMO controls are negatively numbered, therefore the array element number must be a negative value.

Example:

```
MemoText STRING(2000)
Customer FILE,DRIVER('Clarion'),PRE(CUS)
Notes    MEMO(2000)
Record   RECORD
Name     STRING(20)
        . .
        CODE
OPEN(Customer)
SET(Customer)
NEXT(Customer)
ASSERT(~ERRORCODE())
Memotext = Customer{PROP:Text,-1}
```

PROP:Value

An array property of a FILE that sets or returns the data contained in a specified MEMO field (use the WHAT procedure for any other type of field). The array element for PROP:Value is a simple negative number which indicates the -nth MEMO.

Example:

```
Text    STRING(2000)
Number  LONG
Customer FILE,DRIVER('TopSpeed'),PRE(CUS)
Notes   MEMO(2000)
Record  RECORD
Number  LONG,DIM(20)
Name    STRING(20)
      . .
CODE
OPEN(Customer)
SET(Customer)
NEXT(Customer)
ASSERT(~ERRORCODE())
Text    = Customer{PROP:Value,-1}           !Get CUS:Notes contents
```


Appendix D - Error Codes

Trappable Run Time Errors

The following errors can be trapped in code with the ERRORCODE and ERROR procedures. Each error has a code number (returned by the ERRORCODE procedure) and an associated text message (returned by the ERROR procedure) indicating what the problem is.

2 File Not Found

The requested file does not exist in the specified directory.

3 Path Not Found

The directory name specified as part of the path does not exist.

4 Too Many Open Files

The total number of file handles available has been used. Check the FILES= setting in the CONFIG.SYS file, or the user's or network's simultaneous open files setting in a network environment.

5 Access Denied

The file has already been opened by another user for exclusive access, has been left in a locked state, or you do not have network rights to open the file. This error can also occur when no disk space is available.

7 Memory Corrupted

Some unknown memory corruption has occurred.

8 Insufficient Memory

There is not enough unallocated memory left to perform the operation. Closing other applications may free up enough memory. With Btrieve, this indicates that you do not have enough real mode memory left to load BTR32.EXE. IN Win95, loading WBTR32.EXE in WINSTART.BAT can avoid this problem.

15 Invalid Drive

An attempt to read a non-existent disk drive has failed.

30 Entry Not Found

A GET to QUEUE has failed. For GET(Q,*key*), the matching *key* value was not found, and for GET(Q,*pointer*), the *pointer* is out of range.

32 File Is Already Locked

An attempt to LOCK a file has failed because another user has already locked it.

33 Record Not Available

Usually an attempt to read past the end or beginning of file with NEXT or PREVIOUS. May also be posted by PUT or DELETE when no record was read before the attempted PUT or DELETE.

35 Record Not Found

For a GET(File,*key*), the matching *key* field value was not found.

36 Invalid Data File

Some unknown data file corruption has occurred, or the OWNER attribute does not match the password used to encrypt the file.

37 File Not Open

An attempt to perform some operation that requires the file be already open has failed because the file is not open.

38 Invalid Key File

Some unknown key file corruption has occurred.

40 Creates Duplicate Key

An attempt to ADD or PUT a record with key field values that duplicate another existing record in the file has been made to a file with a key that does not allow duplicate entries.

43 Record Is Already Held

An attempt to HOLD a record has failed because another user has already held it.

45 Invalid Filename

The filename does not meet the definition of a valid DOS filename.

46 Key File Must Be Rebuilt

Some unknown key corruption has occurred that requires the BUILD statement to rebuild the key.

47 Invalid Record Declaration

The data file on disk does not match the file's declaration in the .EXE, usually because you have changed the file's definition in the Data Dictionary and have not yet converted the existing data file to the new format. See How do I handle an Error 47

48 Unable To Log Transaction

A transaction logout or pre-image file cannot be written to disk. This usually occurs because no disk space is available, or the user does not have the proper network rights.

52 File Already Open

An attempt to OPEN a file that has already been opened by this user.

54 No Create Attribute

An attempt to execute the CREATE procedure on a file whose declaration does not include the CREATE attribute.

55 File Must Be Shared

An attempt to open a file for exclusive access that must be shared. (Legacy error, no longer used)

56 LOGOUT Already Active

An attempt to issue a second LOGOUT statement while a transaction is already in progress.

57 Invalid Memo File

Some unknown memo file corruption has occurred. For Clarion data files, this could come from a corrupt .MEM file "signature" or pointers to the memo file in the data file that are "out of sync" (usually due to copying files from one location to another and copying the wrong .MEM file).

63 Exclusive Access Required

An attempt to perform a BUILD(file), BUILD(key), EMPTY(file) or PACK(file) was made when the file had not been opened with exclusive access.

64 Sharing Violation

An attempt to perform some action on a file which requires that the file be opened for shared access.

65 Unable To ROLLBACK Transaction

An attempt to ROLLBACK a transaction has failed for some unknown reason.

73 Memo File Missing

An attempt to OPEN a file that has been declared with a MEMO field and the file containing that memo data does not exist.

75 Invalid Field Type Descriptor

Either the type descriptor is corrupt, you have used a *name* that does not exist in GET(Q,*name*), or the file definition is not valid for the file driver. For example, trying to define a LONG field in an xBase file without a matching MEMO field.

76 Invalid Index String

The index *string* passed to BUILD(DynIndex,*string*) was invalid.

77 Unable To Access Index

An attempt to retrieve records using a dynamic index failed because the dynamic index could not be found.

78 Invalid Number Of Parameters

You did not pass the correct number of parameters to a procedure called in an EVALUATE statement.

79 Unsupported Data Type In File

The file driver has detected a field in the file declared with a data type that is not supported by the file system the driver is designed to access.

80 Unsupported File Driver Function

The file driver has detected a file access statement that is not supported. This is frequently an unsupported form (different parameters) of a statement that is supported.

81 Unknown Error Posted

The file driver has detected some error from the backend file system that it cannot get further information about.

88 Invalid Key Length

An attempt to CREATE a Clarion file driver KEY or INDEX with more than 245 characters. Other file drivers can also return this error when their file system key length limits are exceeded.

89 Record Changed By Another Station

The WATCH statement has detected a record on disk that does not match the original version of the record about to be updated in a network situation.

90 File Driver Error

The file driver has detected some other error reported by the file system. You can use the FILEERRORCODE and FILEERROR procedures to determine exactly what native error the file system is reporting.

91 No Logout Active

The COMMIT or ROLLBACK statement has been issued outside of a transaction frame (no LOGOUT statement has been executed).

92 BUILD in Progress

A BUILD statement has been issued and PROP:ProgressEvents has been set to generate events. The statement generating this error is not appropriate to execute during a BUILD process.

93 BUILD Cancelled

The user cancelled the BUILD. This error is set when EVENT:BuildDone is posted.

94 Record Limit Exceeded

The target file has exceeded the record limit. This value is file driver dependant, and can be returned during any attempt to modify a file where the record limit is exceeded.

97 Stream Error

Used during RTF processing.

100 Trigger Error

This error is set whenever a registered file callback method returns FALSE. See CALLBACK

800 Illegal Expression

The EVALUATE procedure has detected an error in the syntax of the expression it is attempting to evaluate.

801 Variable Not Found

The EVALUATE procedure has not found a variable used in the expression it is attempting to evaluate. You must first BIND all variables used in the expression for them to be visible to EVALUATE.

Non-Trappable Run Time Errors

The following errors occur at run time and cannot be trapped with the ERRORCODE or ERROR procedures.

ACCEPT loop requires a window

An ACCEPT loop that has no associated window.

ENDPAGE must only be called for reports

An attempt to execute the ENDPAGE statement when no REPORT is active.

Event posted to a report control

An attempt to POST an event to a control in a REPORT structure.

Metafile record too large in report

A .WMF file is too large to print in the report.

PRINT must only be called for reports

An attempt to PRINT a structure that is not part of a REPORT.

Report is already open

An attempt to OPEN a REPORT that has already been opened and not yet closed.

Too many keystrokes PRESSED

The parameter to the PRESS statement contains too many characters.

Unable to complete operation (system is MODAL)

An attempt to perform an illegal action in a program that has already opened a MODAL window or is processing a modal event.

Unable to create control (system is MODAL)

An attempt to CREATE a control in a program that has already opened a MODAL window or is processing a modal event.

Unable to open APPLICATION (APPLICATION already active)

An attempt to OPEN an APPLICATION in a program that has already opened an MDI application frame window.

Unable to open APPLICATION (system is MODAL)

An attempt to OPEN an APPLICATION in a program that has already opened a MODAL window or is processing a modal event.

Unable to open APPLICATION

A failed attempt to OPEN an APPLICATION.

Unable to open MDI window (No APPLICATION active)

An attempt to OPEN an MDI WINDOW in a program that has not yet opened an MDI APPLICATION frame window.

Unable to open MDI window (system is MODAL)

An attempt to OPEN an MDI WINDOW in a program that has already opened a MODAL window or is processing a modal event.

Unable to open MDI window on APPLICATION's thread

An attempt to OPEN an MDI WINDOW in the same execution thread as the MDI APPLICATION frame window.

Unable to open MDI WINDOW

A failed attempt to OPEN an MDI WINDOW.

Unable to open WINDOW

A failed attempt to OPEN a WINDOW.

Unable to process ACCEPT (system is MODAL)

An attempt to perform an illegal action in a program that has already opened a MODAL window or is processing a modal event.

Unexpected error opening printer device

An unexpected error occurred while attempting to open a printer.

Window is already open

An attempt to OPEN a WINDOW that is already open.

Window is not open

An attempt has been made to perform some action that requires a window be opened first. Usually a property assignment statement.

WSLSTG 755

Indicates that the operating system has failed when asking for more memory. This probably indicates that you have a screen or report structure too complex to generate. Also, ensure that you do not have Clarion set run in its own memory space. The operating system may then limit allocated virtual memory.

Compiler Errors

The compiler generates an error message at exactly the point in the source code where it determines that something has gone wrong. Therefore, the problem is always either right at that point, or somewhere in the code preceding that point. For most error messages, the problem exists right at the point at which it is detected, but some error messages are typically generated by problems that far precede their detection by the compiler, making some "detective work" necessary, along with an understanding of what the compiler is trying to tell you in the error message itself.

Deciphering compiler error messages to determine exactly what syntax error needs to be corrected can be a bit of an arcane science. The major reason for this is that a single (relatively minor) error can create a "cascade effect;" a long list of error messages that all have one root cause. This is typically the case in the situation where there are a very large number of compiler errors reported in the same source module. To handle this, you should correct just the first error reported then re-compile to see how many errors are left (quite often, none). If you have just a couple of errors reported that are widely separated in the source code, it is likely that each is a discrete error and you should correct them all before re-compiling.

Specific Errors

The following error messages occur when the compiler has detected a specific syntax problem and is attempting to alert you to exactly what the problem is so that you may correct it.

Some of the following error messages contain a "%V" token. The compiler substitutes an explicit label indicating what problem is occurring for this token when it generates the error message, which should help point to the cause of the error.

! introduces a comment

This is a common C programmer's error. If you type `IF A != 1 THEN` you get this warning.

Actual value parameter cannot be array

The passed parameter must not be an array.

ADDRESS parameter ambiguous

`ADDRESS(MyLabel)` where *MyLabel* is the label of both a procedure and a data item.

All fields must be declared before JOINS

All PROJECT statements for the file must precede any JOIN statements in the VIEW structure.

Ambiguous label

The field qualification syntax has come up with more than one solution for the label you have supplied.

For example:

```
G   GROUP
S:T SHORT   !Referenced as G:S:T
      END
G:S GROUP
T   SHORT   !Referenced as G:S:T
      END
CODE
G:S:T = 7   !Which are you talking about?
```

Attribute parameter must be QUEUE, QUEUE field or constant string

The parameter must be the label of a previously declared QUEUE structure, a field within a QUEUE structure, or a string constant.

Attribute requires more parameters

You must pass all required parameters to an attribute that takes parameters.

Attribute string must be constant

The parameter must be a string constant, not the label of a variable.

Attribute variable must be global

The parameter must be a variable declared in the PROGRAM module as global data.

Attribute variable must have string type

The parameter must be a variable declared as a STRING, CSTRING, or PSTRING.

BREAK structure must enclose DETAIL

There must be at least one DETAIL structure within nested BREAK structures (at the lowest level).

Calling function as procedure

A Warning that a PROCEDURE which returns a value and does not have the PROC attribute is being called as a PROCEDURE without a return value would be and the return value will be lost.

Cannot call procedure as function

You cannot call a PROCEDURE which does not return a value as the source of an assignment statement or as a parameter.

Cannot declare KEY in a VIEW

A KEY declaration is not valid in a VIEW structure.

Cannot EXIT from here

Only a ROUTINE may contain the EXIT statement.

Cannot GOTO into ROUTINE

The target of GOTO must be the label of an executable code statement within the same procedure or ROUTINE, and may not be the label of a ROUTINE.

Cannot have default parameter here

You may only have a default value on non-omittable integer data type parameters passed by value.

Cannot have initial values with OVER

A variable declaration with the OVER attribute may not also have an initial value parameter.

Cannot have statement here

This happens if the compiler thinks you have tried to define a code label inside the global data section.

Cannot initialize variable reference

A reference variable cannot have an initial value.

Cannot return CSTRING from CLARION function

CSTRING is not a valid return data type for a PROCEDURE written in Clarion (only for functions written in other languages).

Cannot RETURN value from procedure

Only a PROCEDURE prototyped to return a value may contain the RETURN statement with a return value parameter.

CLARION function cannot use RAW or NAME

These attributes are not appropriate for a PROCEDURE written in Clarion (only for functions written in other languages).

DECIMAL has too many places

A DECIMAL or PDECIMAL declaration may only have a maximum of 30 places to the right of the decimal, and the decimal portion must be less than the total length.

DECIMAL too long

A DECIMAL or PDECIMAL declaration may have a maximum length of 31 digits.

Declaration not valid in FILE structure

This data declaration may not be contained within a FILE structure.

Declaration too big

The compiler has detected a PSTRING > 255 or MEMO > 64K in 16 bit, etc.

DLL attribute requires EXTERNAL attribute

The DLL attribute further defines the EXTERNAL attribute and is necessary in 32-bit programs.

Dynamic INDEX must be empty

An attempt to use the 2 parameter form of BUILD on a KEY or INDEX declared with component fields.

Embedded OVER must name field in same structure

The parameter to the OVER attribute must be the label of a previously declared variable in the same structure.

ENCRYPT attribute requires OWNER

The ENCRYPT attribute and OWNER attribute function together.

Entity-parameter cannot be an array

You cannot pass an array of entity parameters (FILE, QUEUE, etc.).

Expected: %V

This is one of the most common errors. The compiler was expecting to find something (one of the items in the list substituted for the %V token) as the next code to compile, but instead found the code at the point in the source that the error is generated.

Expression cannot be picture

You have attempted to use an EQUATE label to a picture token in a place where a picture token is not valid.

Expression cannot have conditional type

An expression is not a numeric value. For example, `MyValue = A > B` is invalid.

Expression must be constant

Variables are not valid in this expression.

Field equate label not defined: %V

The named field equate label has not been previously declared.

Field not found

Using field qualification syntax to reference a field that is not in the parent item. For example, referencing `MyGroup.SomeField` where `SomeField` is not in the `MyGroup` declaration.

Field not found in parent FILE

A JOIN statement must declare all the linking fields between the parent and child files.

Field requires (more) subscripts

This is referencing an array with multiple dimensions, and you must supply an index into each dimension.

FILE must have DRIVER attribute

The DRIVER attribute is required to declare the file system for which the data file is formatted.

FILE must have RECORD structure

It is invalid to declare a FILE which does not contain a RECORD structure.

FILES must have same DRIVER attribute

All files named in a LOGOUT statement must use the same file system.

Function did not return a result

A warning that the implementation of the PROCEDURE prototyped to return a value did not return a result.

Function result is not of correct type

The RETURN statement must return a value consistent with the return data type prototyped in the MAP structure.

Group too big

GROUPs are limited to 64K in 16 bit.

Ignoring EQUATE redefinition: %V

A Warning that the named equate is being ignored. This is really a label-redefined error except that the definition is not thrown away.

Illegal array assignment

An assignment to an array must reference a single element, not the entire array.

Illegal character

A non-valid lexical token. For example, an ASCII 255 in your source.

Illegal data type: %V

The named data type is inappropriate for the structure in which it is placed.

Illegal key component

A KEY has any type of illegal component.

Illegal nesting of window controls

Window controls other than RADIO have been placed within an OPTION structure, or controls other than TAB have been placed directly within a SHEET structure.

Illegal parameter for LIKE

An illegal parameter to a LIKE declaration. For example, LIKE(7).

Illegal parameter type for STRING

An illegal parameter to a STRING declaration. For example, STRING(MyVar) where MyVar is the label of a variable and not an EQUATE.

Illegal reference assignment

A reference variable may only be assigned another reference variable of the same type, or the label of a variable of the type it references.

Illegal return type or attribute

The prototype contains an invalid data type as the return data type (such as *CSTRING).

Illegal target for DO

The target of DO must be the label of a ROUTINE.

Illegal target for GOTO

The target of GOTO must be the label of an executable code statement within the same procedure or ROUTINE, and may not be the label of a ROUTINE.

INCLUDE invalid, expected: %V

The INCLUDE statement's parameter must be a well formed Clarion string. In particular, type conversion is not valid, so INCLUDE('MyFile'&MyValue) is invalid.

INCLUDE misplaced

INCLUDE has to follow a line-break, or a semi-colon (possibly followed by white space).

INCLUDE nested too deep

You can only nest INCLUDEs 3 deep. In other words you can INCLUDE a file that INCLUDEs a file that INCLUDEs a file, but the last file must not INCLUDE anything.

Incompatible assignment types

An attempt to assign between incompatible data types.

Incorrect procedure profile

An attempt to pass a procedure with the wrong prototype as a procedure-parameter.

Indices must be constant

An attempt has been made to have a USE variable that is an array element with variable indices.

Indistinguishable new prototype: %V

A prototype that the compiler cannot uniquely distinguish from a previous prototype using the rules for procedure overloading .

Integer expression expected

The expression must evaluate to an integer.

Invalid BREAK statement

A BREAK that attempts to break to a non-LOOP label or is outside a LOOP or ACCEPT structure.

Invalid CYCLE statement

A CYCLE that attempts to cycle to a non-LOOP label or is outside a LOOP or ACCEPT structure.

Invalid data declaration attribute

An attribute that is inappropriate on the data declaration.

Invalid data type for value parameter

The data type prototyped in the MAP may not be passed by value and must be passed by address. For example, to pass a CSTRING parameter to a Clarion procedure, it may only be prototyped as *CSTRING.

Invalid FILE attribute

An attribute that is inappropriate on a FILE declaration.

Invalid first parameter of ADD

The statement's first parameter is not appropriate.

Invalid first parameter of FREE

The statement's first parameter is not appropriate.

Invalid first parameter of NEXT

The statement's first parameter is not appropriate.

Invalid first parameter of PUT

The statement's first parameter is not appropriate.

Invalid GROUP/QUEUE/RECORD attribute

An attribute that is inappropriate on a GROUP, QUEUE, or RECORD declaration.

Invalid KEY/INDEX attribute

An attribute that is inappropriate on a KEY or INDEX declaration.

Invalid label

A label that contains characters other than letters, numbers, underscore (_), or colon (:), or does not start with a letter or underscore.

Invalid LOOP variable

An attempt to use an illegal data type (DATE, TIME, STRING, etc.) as a LOOP variable.

Invalid MEMBER statement

The parameter to the MEMBER statement is not a string constant or does not reference the PROGRAM module for the current project.

Invalid method invocation syntax

An attempt to use the {} syntax for method invocation on a BLOB or FILE.

Invalid number

A number is required, for example inside the repeat character notation ({} in a string constant.

Invalid OMIT expression

The parameter to the OMIT statement is invalid.

Invalid parameters for attribute

You must pass valid parameters to an attribute that takes them.

Invalid picture token

A picture token that contains inappropriate characters.

Invalid printer control token

A PRINT statement containing a printer control token.

Invalid QUEUE/RECORD attribute

An attribute that is inappropriate on a QUEUE or RECORD declaration.

Invalid SIZE parameter

SIZE(Junk+SomeMoreJunk)

Invalid string (misused <...> or {...})

A string constant contains a single beginning bracket (< or {) without a matching terminating bracket (> or }). These characters must have two together (<< or {{) if intended to be part of the string.

Invalid structure as first parameter

The statement's first parameter is not appropriate.

Invalid structure within property syntax

A structure that is inappropriate in a property assignment statement.

Invalid USE attribute parameter

The parameter is not appropriate for a USE attribute.

Invalid use of PRIVATE data

Attempt to access a PRIVATE data member outside the CLASS module.

Invalid use of PRIVATE procedure

Attempt to call a PRIVATE method outside the CLASS module.

Invalid variable data parameter type

When passing parameters by address, you must pass the same data type as prototyped in the MAP structure.

Invalid WINDOW control

A control that is inappropriate in a WINDOW structure.

ISL error: %V

Contact Technical Support and provide all details of the error message.

KEY must have components

You cannot declare a KEY without naming the component fields that establish the KEY's sort order.

Label duplicated, second used: %V

The named field equate label is used multiple times within the same module and only the last encountered is used in the list of equate labels that may be used within the executable code. Correctable with the third parameter to the USE attribute.

Label in prototype not defined: %V

Using a prototype where one of the data types has not yet been defined.

Label not defined: %V

The named label has not been previously declared.

Mis-placed string slice operator

A string slice that is not the last array index. For example, MyStringArray[3:4,5].

Missing procedure definition: %V

The named procedure is not prototyped in a MAP structure.

Missing virtual function

Compiler bug.

Must be dimensioned variable

This must be an array.

Must be field of a FILE or VIEW

Must be a field that is a member of a FILE or VIEW structure. For example NULL(LocalVariable) with give this error.

Must be FILE or KEY

The parameter to JOIN is not a FILE or KEY label.

Must be reference variable

You can only DISPOSE of a reference variable.

Must be variable

This must be the label of a previously declared variable.

Must have constant string parameter

The parameter must be a string constant, not the label of a variable.

Must RETURN value from function

A PROCEDURE prototyped to return a value must contain the RETURN statement with a return value parameter.

Must specify DECIMAL size

A DECIMAL or PDECIMAL declaration must declare the maximum number of digits it stores.

Must specify identifier

An identifier was required but not supplied.

Must specify print-structure

A PRINT statement may only print a structure in a REPORT.

No matching prototype available

Attempt to define a procedure for which there is no matching prototype in a MAP or CLASS.

Not valid inside structure

A data type is inappropriate for the structure in which it is placed.

OMIT cannot be nested

You are in an OMIT (or COMPILE) that is *not* omitting code and the compiler encounter another OMIT.

OMIT misplaced

OMIT has to follow a line-break, or a semi-colon (possibly followed by white space).

OMIT not terminated: %V

The referenced OMIT parameter was not found before the end of the source module.

Order is MENUBAR, TOOLBAR, Controls

The MENUBAR structure must come before the TOOLBAR, and the TOOLBAR structure must come before the controls in a WINDOW or APPLICATION.

OVER must name variable

The parameter to the OVER attribute must be the label of a previously declared variable.

OVER must not be larger than target variable

The parameter to the OVER attribute must be the label of a previously declared variable that is greater than or equal to the size of the variable being declared OVER it.

OVER not allowed with STATIC or THREAD

A variable declaration with the OVER attribute may not also have the STATIC or THREAD attribute (these must be on the initial declaration).

Parameter cannot be omitted

The procedure call must pass all parameters that have not been prototyped as omissible parameters.

Parameter kind does not match

When passing parameters by address, you must pass the same data type as prototyped in the MAP structure.

Parameter must be picture

This must be a display picture token.

Parameter must be procedure label

This must be the label of a procedure.

Parameter must be report DETAIL label

A PRINT statement may only print a DETAIL structure in a REPORT.

Parameters must have labels

Attempt to define a procedure without using labels on parameters.

Parameter type label ambiguous (CODE or DATA)

You may have a PROCEDURE and data declaration with the same name, but then you cannot use that name in a procedure prototype.

PROCEDURE cannot have return type

If you declare a prototype without a return data type in the MAP, you must create it as a PROCEDURE.

Procedure doesn't belong to module: %V

An attempt to define a procedure that has a prototype that says it belongs in another module.

Procedure in parent CLASS has VIRTUAL mismatch

Virtual methods require the VIRTUAL attribute on the prototypes in both the parent and derived CLASSES.

Prototype is: %V

Attempt to define a procedure with the wrong prototype.

QUEUE/RECORD not valid in GROUP

A GROUP structure may not contain a QUEUE or RECORD structure.

Redefining system intrinsic: %V

A Warning that the named procedure (part of your source code) has the same name as a Clarion run time library procedure and that your procedure will be called instead of the built-in library's.

Routine label duplicated

The label of a ROUTINE statement has been previously used on another statement.

Routine not defined: %V

The named ROUTINE does not exist.

SECTION duplicated: %V

The named SECTION exists twice in the INCLUDE file.

SECTION not found: %V

The named SECTION does not exist in the INCLUDE file.

Statement label duplicated

Two lines of executable source code have the same label.

Statement must have label

The statement (such as a ROUTINE or PROCEDURE statement) must have a label.

String not terminated

A string constant without a terminating single quote (').

Subscript out of range

An attempt to reference an array element beyond the valid number of elements dimensioned in the data declaration.

Too few indices

This is referencing an array with multiple dimensions, and you must supply an index into each dimension.

Too few parameters

The procedure call must pass all parameters that have not been prototyped as omissible parameters.

Too many indices

This is referencing an array and you are supplying too many indexes into the dimensions.

Too many parameters

The procedure call may not pass more parameters than have been prototyped.

Unable to verify validity of OVER attribute

A Warning that you are declaring a variable OVER a passed parameter and the data types may not match at run time.

Unknown attribute: %V

The named attribute is not part of the Clarion language.

Unknown function label

The PROCEDURE has not been previously prototyped in a MAP structure.

Unknown identifier

The label has not been previously declared.

Unknown identifier: %V

The named identifier has not been previously declared.

Unknown key component: %V

The named key component does not exist within the FILE structure.

Unknown procedure label

The PROCEDURE has not been previously prototyped in a MAP structure.

UNTIL/WHILE illegal here

Attempt to use UNTIL or WHILE to terminate a LOOP structure that is already terminated.

Value-parameter cannot be an array

You cannot pass an array as a value-parameter.

Value requires (more) subscripts

This is referencing an array with multiple dimensions, and you must supply an index into each dimension.

Variable expected This must be the label of a previously declared variable.

Variable-size must be constant

The variable declaration must contain a constant expression for its size parameter.

VIRTUAL illegal outside of CLASS structure

You can only use the VIRTUAL attribute on prototypes in a CLASS structure, not in a MAP.

Wrong number of parameters

The procedure call must pass all parameters that have not been prototyped as omissible parameters.

Wrong number of subscripts

An attempt to access a multi-dimensioned array without providing an element number for each dimension.

For example:

```
MyShort SHORT,DIM(8,2) !Two-dimensional array
CODE
MyValue = MyShort[7] !Wrong number of subscripts error
```

Unknown errors

These are errors that should never happen and are only to give the compiler writer a clue as to what is wrong. Report the problem immediately to SoftVelocity together with the source file that generated the error.

- Inconsistent scanner initialization
- Unknown operator
- Unknown expression type
- Unknown expression kind
- Unknown variable context
- Unknown parameter kind
- Unknown assignment operator
- Unknown variable type
- Unknown case type
- Unknown equate type
- Unknown string kind
- Unknown picture type
- Unknown descriptor type
- Unknown initializer type
- Unknown designator kind
- Unknown structure field
- Unknown formal entity
- Type descriptor not static
- Unknown clear type
- Unknown simple formal type
- Out of attribute space
- Unknown label/routine
- Unknown special identifier
- Value not static
- Unknown static label
- Unknown screen structure kind
- Corrupt pragma string

Old symbol non-NIL

Not implemented yet

String not CCST

Appendix E - Legacy Statements

Legacy Statements

All the statements listed in this Appendix are valid only for compatibility with previous versions of Clarion. They are all subject to complete removal in future releases of Clarion and so, should not be used.

BOF (return beginning of file)

EOF (return end of file)

FUNCTION (define a function)

POINTER (return relative record position)

SHARE (open data file for shared access)

BOF (return beginning of file)

BOF(*file*)

BOF Flags the beginning of the FILE during sequential processing.

file The label of a FILE declaration.

The **BOF** procedure returns a non-zero value (true) when the first record in relative file sequence has been read by PREVIOUS or passed by SKIP. Otherwise, the return value is zero (false).

The BOF procedure is not supported by all file drivers, and can be very inefficient even if supported (check the driver documentation). Therefore, for efficiency and guaranteed file system support it is not recommended to use this procedure. Instead, check the ERRORCODE() procedure after each disk read to detect an attempt to read past the beginning of the file.

The BOF procedure was most often used as an UNTIL condition evaluated at the top of a LOOP, so BOF returns true after the last record has been read and processed in reverse order (using PREVIOUS).

Return Data Type: LONG

Example:

!Not recommended, but still supported for backward compatibility:

```
SET(Trn:DateKey)           !End/Beginning of file in keyed sequence
LOOP UNTIL BOF(Trans)      !Process file backwards
  PREVIOUS(Trans)         ! read a record sequentially
  IF ERRORCODE() THEN STOP(ERROR()).
  DO LastInFirstOut       ! call last in first out routine
END
```

!Recommended as most efficient code for use with all file drivers:

```
SET(Trn:DateKey)           !End/Beginning of file in keyed sequence
LOOP                       !Process file backwards
  PREVIOUS(Trans)         ! read a record sequentially
  IF ERRORCODE() THEN BREAK. !Break loop at attempt to read past beginning
  DO LastInFirstOut       ! call last in first out routine
END
```

See Also:

ERRORCODE

EOF (return end of file)

EOF(*file*)

EOF Flags the end of the FILE during sequential processing.

file The label of a FILE declaration.

The **EOF** procedure returns a non-zero value (true) when the last record in relative file sequence has been read by NEXT or passed by SKIP. Otherwise, the return value is zero (false).

The EOF procedure is not supported by all file drivers, and can be very inefficient even if supported (check the driver documentation). Therefore, for efficiency and guaranteed file system support it is not recommended to use this procedure. Instead, check the ERRORCODE() procedure after each disk read to detect an attempt to read past the end of the file.

The EOF procedure was most often used as an UNTIL condition at the top of a LOOP, so EOF returns true after the last record has been read and processed.

Return Data Type: LONG

Example:

```
!Not recommended, and still available for backward compatibility:
SET(Trn:DateKey)                !Beginning of file in keyed sequence
LOOP UNTIL EOF(Trans)           !Process all records
  NEXT(Trans)                   ! read a record sequentially
  IF ERRORCODE() THEN STOP(ERROR()).
  DO LastInFirstOut             ! call last in first out routine
END
```

```
!Recommended for use with all file drivers:
SET(Trn:DateKey)                !Beginning of file in keyed sequence
LOOP                             !Process all records
  NEXT(Trans)                   ! read a record sequentially
  IF ERRORCODE() THEN BREAK. !Break loop on attempt to read past end of file
  DO LastInFirstOut             ! call last in first out routine
END
```

See Also:

ERRORCODE

FUNCTION (define a function)

label **FUNCTION** [(*parameter list*)]

local data

CODE

statements

RETURN(*value*)

FUNCTION is a statement which once defined a PROCEDURE prototyped to return a value (referred to as a function in some other programming languages). The FUNCTION keyword has been replaced by the PROCEDURE statement and is now a synonym for PROCEDURE in all cases.

Example:

```

PROGRAM
MAP
FullName FUNCTION(STRING Last,STRING First,<STRING Init>),STRING
                                !Function prototype with parameters
DayString FUNCTION,STRING      !Function prototype without parameters
END
TodayString STRING(3)
CODE
TodayString = DayString()      !Function call without parameters
                                ! the () is required for a function

FullName FUNCTION(STRING Last, STRING First,STRING Init)      !Full name function
CODE                                                            !Begin executable code section
IF OMITTED(3) OR Init =                                         !If no middle initial
RETURN(CLIP(First) &      & Last) ! return full name
ELSE                                                            !Otherwise
RETURN(CLIP(First) &      & Init & . & Last)                    ! return full name
END

DayString FUNCTION      !Day string function
ReturnString STRING(9),AUTO !Uninitialized local stack variable
CODE                    !Begin executable code section
RETURN(CHOOSE(TODAY()%7)+1,Sun,Mon,Tue,Wed,Thu,Fri,Sat))

```

See Also:

PROCEDURE

POINTER (return relative record position)

```
POINTER(      | file  | )  
             | key   |
```

POINTER	Returns relative record position.
<i>file</i>	The label of a FILE declaration. This specifies physical record order within the file.
<i>key</i>	The label of a KEY or INDEX declaration. This specifies the entry order within the KEY or INDEX file.

POINTER returns the relative record position within the data file (in *file* sequence), or the relative record position within the KEY or INDEX file (in *key* sequence) of the last record accessed.

The value returned by the POINTER procedure is dependent on the file driver. It may be a record number, the relative byte position within the file, or some other kind of seek position within the file.

The POINTER procedure is not supported by all file drivers. Therefore it should be used only when you know the file system supports it and you will not be changing file systems in the future. The preferred method of record positioning that is designed to work across all file systems is the POSITION procedure with RESET and REGET.

Return Data Type: LONG

Example:

```
SavePtr# = POINTER(Customer)  !Save file pointer
```

See Also:

POSITION

SHARE (open data file for shared access)

SHARE(*file* [,*access mode*])

SHARE	Opens a FILE structure for processing.
<i>file</i>	The label of a FILE declaration.
<i>access mode</i>	A numeric constant, variable, or expression which determines the level of access granted to both the user opening the file, and other users in a multi-user system. If omitted, the default value is 42h (Read/Write, Deny None).

The **SHARE** statement opens a FILE structure for processing and sets the *access mode*. The SHARE statement is exactly the same as the OPEN statement, with the exception of the default value of *access mode*.

The *access mode* is a bitmap which tells the operating system what access to grant the user opening the file and what access to deny to others using the file.

The actual values for each access level are:

	<u>Dec.</u>	<u>Hex.</u>	<u>Access</u>
User Access:	0	0h	Read Only
	1	1h	Write Only
	2	2h	Read/Write
Others Access:	0	0h	Any Access (FCB compatibility mode)
	16	10h	Deny All
	32	20h	Deny Write
	48	30h	Deny Read
	64	40h	Deny None

Errors Posted: The same set of errors that may be posted by OPEN

Example:

```

ReadOnly    EQUATE(0)                !Access mode equates
WriteOnly   EQUATE(1)
ReadWrite   EQUATE(2)
DenyAll     EQUATE(10h)
DenyWrite   EQUATE(20h)
DenyRead    EQUATE(30h)
DenyNone    EQUATE(40h)
CODE
  SHARE(Master,ReadOnly+DenyWrite)  !Open read only mode

```

See Also: OPEN

Index:

- ! Comments 39
- .DLL 243
- .ENV 209, 759
- .INI
 - GETINI 727
 - PUTINI 828
- .OCX Control Properties 971
- := 561
- ? (field equate labels) 39, 299
- ? debug 39
- ABCDIIMode 237
- ABS 608
- ABS (return absolute value) 608
- ABSOLUTE 391
- ABSOLUTE (set fixed-position printing) .. 391
- ACCEPT 579, 580
- AcceptAll 1009
- AcceptAll mode 854, 855
- Accepted 1003
- ACCEPTED 609
- ACCEPTED (return control just completed)
 - 609
- Access Mode 791, 793
- ACOS 610
- ACOS (return arccosine) 610
- ACTIVE 1010
- ADD 611
- ADD (add an entry) 611
- Addition operator 538
- ADDRESS 615
- ADDRESS (return memory address) 615
- AGE 616
- AGE (return age from base date) 616
- Alarm (BEEP) 627
- AlertKey 997
- Algebraic Order of Operation 537
- ALIAS 619
- ALIAS (set alternate keycode) 619
- ALL 620
- ALL (return repeated characters) 620
- allocate heap memory 156
- Allocation
 - memory
 - Dynamic 153
 - Static 153
- ALLTRIM 653
- ALONE 392
- ALONE (set to print without page header footer
 - or form) 392
- Alphanumeric
 - CSTRING 130
 - PSTRING 133
 - STRING 126
- AlwaysDrop 1010
- AND 540
- ANGLE 395
- ANGLE (set control display or print angle)
 - 395
- ANY 141
- ANY (any simple data type) 141
- APPEND 621
- APPEND (add a new file record) 621
- AppInstance 1011
- APPLICATION 277, 278, 279, 280, 281, 282
- APPLICATION (declare an MDI frame window)
 - 277
- Application modal 283
- Application Modal Windows 476, 478
- Application windows 297
- ARC 622
- Arithmetic Operator
 - Addition 538
 - Division 538
 - Exponentiation 538
 - Modulus 538
 - Multiplication 538
 - Subtraction 538
- Arithmetic Operators 538, 543
- Arithmetic overflow 159, 566
- Array
 - DIM 235
 - subscript 537, 549
- array elements
 - field equate labels 299
- ASCII Character Codes 542
- ASIN 624
- ASIN (arcsine) 624
- ASIN (return arcsine) 624
- ASK 623

ASK (get one keystroke).....	623	BEGIN (define code structure).....	89
ASSERT	87	BFLOAT4	118
ASSERT (set assumption for debugging) .	87	BFLOAT4 (four-byte signed floating point)	
AssertHook	1011	118
AssertHook2	1012	BFLOAT8	120
Assignment Statements		BFLOAT8 (eight-byte signed floating point)	
CLEAR	565	120
Deep	561	Binary	
Operating	560	Numeric constant	541
Reference	563, 564	BINARY	232
Simple	559	BINARY (memo contains binary data).....	232
ASTRING	128, 129	Binary Coded Decimal (BCD)	569
ASTRING (atomic string)	128	BIND	550
AT	396	BIND (declare runtime expression string	
AT (set position and size)	396	variable).....	550
ATAN	625	BINDABLE	233
ATAN (arctangent).....	625	BINDABLE (set runtime expression string	
ATAN (return arctangent)	625	variables).....	233
Attribute Equates	389	BINEXPRESSION	552
Attribute Property Equates	389	Bit manipulation	
AUTO	231, 399	BXOR	637
AUTO (set USE variable automatic re-		Bit Manipulation	626, 629, 631
display).....	399	BLANK	628
AUTO (uninitialized local variable)	231	BLANK (erase graphics)	628
automatic constructor	174	Blank when zero	159
Automatic Conversion of Data Types	566	BLOB	193, 194, 195
automatic destructor	174	BLOB (declare a variable-length field).....	193
AutoPaper	1013	BOF	1128
AUTOSIZE	400	BOF (return beginning of file)	1128
AUTOSIZE (set OLE object resizing)	400	Boolean operators	540
B (blank when zero).....	159	BOR	629
BAND	626	BOR (Bitwise OR).....	629
BAND (Bitwise AND)	626	BOR (return bitwise OR).....	629
BAND (return bitwise AND)	626	BOX	321, 322, 630
Base 10 logarithm.....	762	BOX (declare a box control)	321
Base Data Types	567	BOX (draw a rectangle)	630
Base numbers		BOXED	403
Binary	541	BOXED (set controls group border).....	403
Decimal	541	BREAK	307, 591
Hexadecimal	541	BREAK (declare group break structure) ..	307
Octal.....	541	BREAK (immediately leave loop).....	591
Base Types	567, 568	BreakVar	1013
BCD	569	BSHIFT	631
BCD Operations and Procedures	569	BSHIFT (Bitwise SHIFT).....	631
BEEP	627	BSHIFT (return shifted bits)	631
BEEP (sound tone on speaker).....	627	Btrieve	
BEGIN.....	89, 584	DATE	137

- LSTRING..... 133
- TIME..... 139
- ZSTRING 130
- Buffer 1014
- BUFFER 632
- BUFFER (set record paging) 632
- BuildDone 997
- BuildKey..... 997
- Built-in variables
 - PRINTER 819
 - TARGET 881, 882
- Built-in Variables..... 546, 547
- BUTTON 323, 324, 325
- BUTTON (declare a pushbutton control) .323
- BXOR (Bitwise eXclusive OR)..... 637
- BXOR (return bitwise exclusive OR) 637
- BY 588, 589, 590
- BYTES 638
- BYTES (return size in bytes) 638
- C
 - PASCAL (parameter passing conventions) 640
- CALL..... 640
- Callback 973, 974, 976, 977
- CALLBACK..... 641
- Calling OLE Object Methods 978
- CAP 404
 - UPR (set case)..... 404
- CASE 582, 583
- CASE (selective execution structure) 582
- Case insensitive key 258
- Case insensitive report break 481
- CENTER 405, 467, 644
- CENTER (return centered string) 644
- CENTER (set centered window position) 405
- CENTERED 406
- CENTERED (set centered image)..... 406
- CHAIN..... 645
- CHAIN (execute another program)..... 645
- CHANGE 646
- Chapter Organization 24
- Character String
 - CSTRING 130
 - PSTRING 133
 - STRING..... 126
- CHECK 326, 327, 407
- CHECK (declare a checkbox control)..... 326
- CHECK (set on/off ITEM) 407
- Checked..... 1014
- Child..... 1015
- ChildIndex..... 1015
- CHOICE 648
- ChoiceFEQ 1016
- CHOOSE 649
- CHORD..... 651
- CHR 652
- CHR (return character from ASCII) 652
- CLAAMPM 209
- CLABUTTON 209
- CLACASE 211
- CLACHARSET 209, 210, 211
- CLACOLSEQ..... 209
- CLADIGRAPH 209
- CLAMON..... 209
- CLAMONTH..... 209
- CLAMSG 211
- Clarion Keycodes..... 31
- Clarion standard date 29
- Clarion standard time..... 30
- CLASS 174
 - LINK 252
 - MODULE 253
- CLASS (object declaration) 174
- CLASS Variables
 - PRIVATE 268
 - PROTECTED 269
- CLEAR 565
- CLEAR (clear a variable) 565
- clear the reference
 - NULL 563, 564
- ClientHandle 1016
- ClientWndProc..... 1017
- CLIP 408, 653
- CLIP (return string without trailing spaces)
 - 653
- CLIP (set OLE object clipping)..... 408
- ClipBits..... 1018
- CLIPBOARD 654
- CLIPBOARD (return windows clipboard contents)..... 654
- CLOCK..... 656
- CLOCK (return system time) 656
- CLONE(duplicate existing control) 657
- CLOSE..... 659
- CLOSE (close a data structure)..... 659

CloseWindow.....	997	CONTENTS (return contents of USE	
CODE	51	variable).....	664
CODE (begin executable statements).....	51	Contracted	1003
collating sequence		Contracting.....	1003
KEY	190	Control Fields.....	298
Collating sequence		Control menu	515
SORT (QUEUE).....	890	Control Numbering.....	299
COLORDIALOG	661	Control Structures.....	582
COLORDIALOG (return chosen color)....	661	Conventions and Symbols.....	26
ColorDialogHook	1019	Conversion	
colored list box fields.....	433	date	
colors in list boxes	350	DEFORMAT	675
COLUMN	412	Conversion of Data Types	566
COLUMN (set list box highlight bar).....	412	convert ANSI strings to ASCII.....	665
ColumnResize	1003	convert ASCII strings to ANSI.....	666
COMBO	331	CONVERTANSITOOEM.....	665
COMBO (declare an entry/list control)	329	CONVERTANSITOOEM (convert ANSI	
COMMAND.....	662	strings to ASCII)	665
COMMAND (return command line)	662	CONVERTOEMTOANSI.....	666
Command line		CONVERTOEMTOANSI (convert ASCII	
COMMAND	662	strings to ANSI)	666
SETCOMMAND	865	Cooperative multi-tasking	932
Comments ! symbol.....	39	COPY	667
COMMIT	663	COPY (copy a file)	667
COMMIT (terminate successful transaction)		Corrupt pragma string.....	1124
.....	663	COS	668
Commit boundaries	764	COS (cosine of angle)	668
Comparison Operators	540	CREATE	234, 415, 669, 670, 671
COMPATIBILITY	414	CREATE (allow data file creation)	234
COMPATIBILITY (set OLE control		CREATE (create an empty data file)	669
compatibility)	414	CREATE (create OLE control object)	415
COMPILE.....	90, 91	CREATE (return new control created)....	670
COMPILE (specify source to compile)	90	Credit (CR) pictures	159
Compiler Directives	87	credit pictures	159
Compiler Error Messages.....	1106	CSTRING.....	130
Compiler Errors	1106	CSTRING (fixed-length null terminated	
Completed	997, 1081	string)	130
Concatenation	543	Currency Pictures	159
Concatenation Operator	539	Current Target	301
Conditional Operators.....	540	CurrentKey.....	1082
ConnectString.....	1081	CURSOR	416
Constants		CURSOR (set mouse cursor type)	416
Numeric Constants	541	CYCLE	592, 593
String Constant	542	CYCLE (go to top of loop).....	592
constructor.....	179	DATA	52
container windows	957	DATA (begin routine local data section)....	52
CONTENTS.....	664	Data Declaration Sections	154

Data Type Conversion.....	570	DDEEXECUTE (send command to DDE server)	944
Data Type Conversion Rules.....	566	DDEITEM.....	946
Data Types		DDEITEM (return server item)	946
ANY	141	DDEMode	1019
BFLOAT4	118	DDEpoke.....	998
BFLOAT8	120	DDEPOKE	947
CSTRING	130	DDEPOKE (send unsolicited data to DDE server)	947
DATE.....	137	DDEQUERY.....	949
DECIMAL	122	DDEQUERY (return registered DDE servers)	949
GROUP	171	DDERead	950
LIKE	144	DDERead (get data from DDE server) ..	950
PDECIMAL.....	124	DDErequest	998
PSTRING	133	DDESERVER	952
REAL.....	116	DDESERVER (return DDE server channel)	952
SREAL	114	DDETimeOut.....	1020
STRING.....	126	DDETOPIC	953
TIME.....	139	DDETOPIC (return server topic).....	953
UNSIGNED	113	DDEVALUE.....	954
Date		DDEVALUE (return data value sent to server)	954
Standard Date	29	DDEWRITE.....	955
DATE	137, 673	DDEWRITE (provide data to DDE client)	955
DATE (four-byte date)	137	de-allocate heap memory	157
DATE (return standard date)	673	Debit (DB) pictures	159
Date conversion		debit pictures	159
DEFORMAT	675	Debug ? symbol.....	39
Date Pictures	164	Decimal	
DAY	674	Numeric Constant	541
DAY (return day of month).....	674	DECIMAL	122, 467
Day of the week.....	29	DECIMAL (signed packed decimal).....	122
DDE	935, 998	Decimal Arithmetic.....	569
DDE Events	936, 998	Declaration and Statement Labels	34
DDE Overview	935	Deep Assignment	561
DDEACKNOWLEDGE.....	938	Deep Assignment Statements	561
DDEadvise.....	998	DEFAULT.....	417
DDEAPP	940	DEFAULT (set enter key button)	417
DDEAPP (return server application).....	940	DeferMove	1020
DDECHANNEL	941	DEFORMAT.....	675
DDECHANNEL (return DDE channel number).....	941	DEFORMAT (return unformatted numbers from string)	675
DDECLIENT	942	DELAY	418
DDECLOSE	943	DELAY (set repeat button delay).....	418
DDECLOSE (terminate DDE server link)	943	DELETE	676, 677, 678, 840
DDEclosed.....	998		
DDEdata	998		
DDEexecute	998		
DDEEXECUTE	944		

DELETE (delete a record)	676	Dollar sign	159, 160
Delete a file (REMOVE).....	840	DOUBLE	423
DERIVED (prevent function overloading) ..	72	NOFRAME	
Derived CLASSes (Inheritance)	175	RESIZE (set window border)	423
Destination variable	559, 560, 561, 563	Double-precision real	116, 120
DESTROY	680	DOWN.....	525
DESTROY (remove a control)	680	Drag	1003
destructor.....	179	Dragging	1003
DETAIL	308	DRAGID	687
DETAIL (report detail line structure)	308	DRIVER	239
Details.....	1083	DRIVER (specify data file type)	239
Dialog boxes	297	DriverLogoutAlias.....	1083
dialog units	285, 396, 398, 433, 435, 466, 491, 509	Drop	1003
DIM	235	DROP.....	425
DIM (set array dimensions)	235	DROP (set list box behavior)	425
DIRECTORY	682, 683	DROPID	689
DIRECTORY (get file directory)	682	DroppedDown	1004
DISABLE.....	419, 685	DroppingDown	1004
DISABLE (dim a control)	685	DUP	240
DISABLE (set control dimmed at open) ..	419	DUP (allow duplicate KEY entries)	240
Dispatch interface.....	966	DUPLICATE.....	691
DISPLAY.....	686	DUPLICATE (check for duplicate key entries)	691
DISPLAY (write USE variables to screen)	686	Dynamic Data	153
DISPOSE.....	157	Dynamic Data Exchange	935
DISPOSE (de-allocate heap memory)	157	dynamic INDEX	189
Division operator.....	538	Dynamic index	
DLL	237, 238, 242	BUILD	634, 635, 636
DLL (set procedure defined externally in .DLL)	73	Edit.....	1022
DLL (set variable defined externally in .DLL)	237	ELLIPSE	334, 335, 692
dll_mode	237	ELLIPSE (declare an ellipse control)	334
DO	594	ELLIPSE (draw an ellipse).....	692
DO (call a ROUTINE)	594	ELSE.....	582, 584, 585, 586, 587
DOCK	420	ELSIF	586, 587
DOCK (set dockable toolbox window)	420	EMPTY.....	693
Docked.....	998	EMPTY (empty a data file).....	693
DOCKED	421	ENABLE	694
DOCKED (set dockable toolbox window docked at open)	421	ENABLE (re-activate dimmed control).....	694
DOCUMENT	422	Enabled.....	1024
DOCUMENT (create OLE control object from file)	422	Encapsulation	174
Document windows	297	ENCRYPT.....	241
Documentation Conventions and Symbols26		ENCRYPT (encrypt data file).....	241
		END	55
		END (terminate a structure).....	55
		ENDPAGE	695
		ENDPAGE (force page overflow)	695
		ENTRY	336, 337, 338

ENTRY (declare a data entry control)	336	Contracted	1003
ENV	209	Contracting	1003
ENV file	759	Docked	998
Environment Files	209	Drag	1003
EOF	1129	Dragging	1003
EOF (return end of file)	1129	Drop	1003
EQUATE	93	DroppedDown	1004
EQUATE (assign label)	93	DroppingDown	1004
ERASE	697	Expanded	1004
ERASE (clear screen control and USE variables)	697	Expanding	1004
ERROR	698	GainFocus	999
ERROR (return error message)	698	Iconize	999
Error Codes	1097	Iconized	999
Error Information	820	Locate	1004
Error messages		LoseFocus	999
Compiler	1106	Maximize	459, 999
Run time	1097, 1103	Maximized	459, 460, 999
Error Messages	1106	MouseDown	1005
ERRORCODE	699	MouseIn	1005
ERRORCODE (return error code number)	699	MouseMove	1005
ERRORFILE	700	MouseOut	1005
ERRORFILE (return error filename)	700	MouseUp	1005
ERRORLEVEL	732, 853	Move	999
EVALUATE	553	Moved	999
EVALUATE (return runtime expression string result)	553	NewSelection	1006
Evaluations		OpenWindow	1000
logical	540	PageDown	1006
Event	998	PageUp	1006
DDEadvise	998	PreAlertKey	1000, 1006
DDEclosed	998	Rejected	836, 1006
DDEdata	998	Restore	1000
DDEexecute	998	Restored	460, 1000
DDEpoke	998	Resume	1000
DDErequest	998	ScrollBottom	1006
EVENT	1005	ScrollDown	1006
Accepted	1003	ScrollDrag	1007
AlertKey	997, 1003	ScrollTop	1007
BuildDone	997	ScrollTrack	1007
BuildFile	997	ScrollUp	1007
BuildKey	997	Selected	1007
CloseDown	997	Size	1000
CloseWindow	997	Sized	1000
ColumnResize	1003	Suspend	1001
Completed	997	TabChanging	1007
		Timer	1001
		Undocked	998
		EVENT:Notify	1000

Events.....	995	FileDialogHook	1026
EventsWaiting.....	1026	FILEERROR	711
Exclude null key entries.....	260	FILEERROR (return file driver error message).....	711
Exclusive Access.....	791	FILEERRORCODE	712
EXECUTE.....	584, 585	FILEERRORCODE (return file driver error code number)	712
BEGIN.....	585	Files	1085
EXECUTE (statement execution structure)	584	FILEs with the EXTERNAL attribute	243
ExeVersion	1026	FILTER.....	245
EXISTS.....	702	FILTER (set view filter expression).....	245
EXISTS (return file existence)	702	FIRST.....	428
EXIT.....	595	LAST (set MENU or ITEM position)	428
EXIT (leave a ROUTINE)	595	FIRSTFIELD	713
Expanded.....	1004	FIRSTFIELD (return first window control).....	713
Expanding.....	1004	FLAT	429
Exponentiation operator	538	FLAT (set flat buttons)	429
Expression Evaluation	537	Floating Point	
Expression Strings.....	549	Double Precision	116, 120
Expressions	537	Single Precision	114, 118
Evaluation Precedence.....	537	floating tabs.....	484
Logical Expressions	545	FLUSH	714
Numeric Expressions	543	FLUSH (flush buffers).....	714
Runtime.....	549	FlushPageNumFunc.....	1027
String Expressions	544	FlushPreview	1028
EXTEND	413	FOCUS	715
EXTERNAL.....	242, 243	FOCUS (return control with focus)	715
EXTERNAL (set defined externally)	242	Follows.....	1029
FatalErrorHook	1026	FONT	430
FetchSize.....	1083	FONT (set default font)	430
Field.....	1084	FONTDIALOG	716
FIELD.....	703	FONTDIALOGA	717
Field Completion Keys		FontDialogHook.....	1029
ALERT.....	617, 618	FOOTER	310
Field Equate Labels.....	299, 300	FOOTER (page or group footer structure)	310
Field Qualification	36	FORM	312
Field-independent events	995, 996	FORM (page layout structure)	312
Fields (controls)	298	FORMAT.....	433, 718
FieldsFile	1084	FORMAT (return formatted numbers into a picture)	718
Field-specific events	995, 1002	FORMAT (set LIST or COMBO layout)	433
File.....	1085	FORMAT() List Box Mouse Click Properties	445
FILE	186, 187, 188	FORMAT() Other List Box Properties	443
FILE (declare a data file structure)	186	FORMAT() Runtime Properties	438
file access.....	791	FORMAT() Style Properties.....	441
file directory	682		
file existence	702		
FILE Structure Properties	198, 199		
FILEDIALOG	704, 705, 706		

- formatting..... 159
- forward reference 150, 151
- FORWARDKEY (pass keystrokes to control
..... 719
- FREE 720
- FREE (delete all entries) 720
- FREESTATE 721
- FREESTATE (free resources) 721
- FROM 448
- FROM (set listbox data source)..... 448
- FULL 450
- FULL (set full-screen) 450
- FUNCTION 1130
- FUNCTION (define a function) 1130
- GainFocus 999
- GET 723, 724, 725
- GETFONT..... 726
- GETINI..... 727
- GETINI (return INI file entry)..... 727
- GETNULLS(get the NULL state of a table)
..... 728
- GETREG..... 730
- GETSTATE (return current state of data file)
..... 731
- Global
 - Local
 - Static
 - and Dynamic..... 153
- Global data 154
- Global menu 290, 291
- Global tools..... 294, 295
- GlobalHelp 1086
- GOTO 596
- GOTO (go to a label) 596
- Graphics Coordinates 302
- Graphics Overview 301
- GRAY..... 451
- GRAY (set 3-D look background) 451
- GRID 452
- GRID (set list grid-line display color) 452
- GROUP..... 171, 340, 341
- GROUP (compound data structure) 171
- GROUP (declare a group of controls) 340
- HALT 732
- HALT (exit program) 732
- HaltHook 1030
- handle 1016
- Handle..... 1030
- HEADER 313
- HEADER (page or group header structure)
..... 313
- HeaderHeight..... 1031
- Held..... 1086
- HELP..... 733
- Hexadecimal (numeric constant) 541
- HIDE 453, 734
- HIDE (blank a control) 734
- HIDE (set control hidden) 453
- HLP 454
- HLP (set on-line help identifier) 454
- HOLD 735, 736
- HOLD (exclusive record access) 735
- HScrollPos 1032
- Iconize..... 999
- Iconized..... 999
- IconList..... 1033
- icons in list boxes..... 350, 433
- icons in list fields 433
- IDLE 737
- IDLE (arm periodic procedure) 737
- IF 586, 587
- IF (conditional execution structure) 586
- IMAGE 343, 344, 738
- IMAGE (declare a graphic image control) 343
- IMAGE (draw a graphic image) 738
- ImageBits 1035
- ImageBlob..... 1036
- IMPLEMENTS(add methods to a CLASS)
..... 248
- Implicit String Arrays 136
- Implicit String Arrays and String Slicing... 136
- Implicit variables
 - LONG 147, 148
 - REAL 147, 148
 - STRING(32) 147, 148
- Implicit Variables..... 147
- INCLUDE 92
- INCLUDE (compile code in another file).... 92
- INCOMPLETE 739
- INCOMPLETE (return empty REQ control)
..... 739
- Inconsistent scanner initialization 1124
- Inheritance 174

INI	
GETINI	727
PUTINI	828
InitAStringHook.....	1036
INLIST.....	740
INLIST (return entry in list)	740
inner join	251
In-place activation.....	958
Input Focus	298
INRANGE	741
INRANGE (check number within range) ..	741
INS.....	461
OVR (set typing mode)	461
INSTANCE.....	742, 743
Instantiation	177
INSTRING.....	744
INSTRING (return substring position) ..	744
INTERFACE	184, 185
INTERFACE (class behavior definition) .	184, 249
Intermediate value	537, 538
Internationalization	209
InToolBar.....	1036
Introduction.....	23
ISALPHA.....	747
ISALPHA (return alphabetic character) ..	747
ISLOWER	748
ISLOWER (return lower case character) .	748
ISSTRING.....	749
ISSTRING (return field string type or not)	749
ISUPPER.....	750
ISUPPER (return upper case character) .	750
ITEM	345, 346, 347
ITEM (declare a menu item).....	345
ITEMIZE.....	94, 95
ITEMIZE (enumeration data structure)	94
Items	1037
JOIN.....	217, 218, 219, 462
JOIN (declare a "join" operation).....	217
JOIN (set joined TAB scroll buttons)	462
KEY.....	190, 463
KEY (declare dynamic file access index)	190
KEY (set execution keycode)	463
KEYBOARD.....	751
KEYBOARD (return keystroke waiting) ...	751
Keyboard Functions.....	751
Keyboard Procedures.....	619
KEYCHAR.....	752
KEYCHAR (return ASCII code)	752
KEYCODE	753
KEYCODE (return last keycode)	753
Keycode EQUATE Labels	31
Keycodes	31
KEYCODES.EQU	31
Key-in Pictures.....	168
Key-in Template Pictures.....	168
KEYSTATE	754
KEYSTATE (return keyboard status).....	754
KEYWORD	
Syntax Diagram.....	27
KEYWORD (short description of intended	
use)	28
label	34
LANDSCAPE	464
LANDSCAPE (set page orientation).....	464
LAST.....	428
LASTFIELD.....	755
LASTFIELD (return last window control) .	755
LAYOUT.....	465
LazyDisplay.....	1037
Leading zeroes	159
LEFT	467, 756
RIGHT	
CENTER	
DECIMAL (set justification).....	467
LEFT (return left justified string)	756
Legacy Statements	1127
LEN	757
LEN (return length of string)	757
LFNSupport.....	1038
LibHook.....	1038
LibVersion.....	1039
LIKE	144, 145
LIKE (inherited data type)	144
Line	1039
LINE	758
LINE (declare a line control)	348
LINE declaration	348
LineCount.....	1039
LineHeight.....	1040
LINewidth	469
LINewidth (set line thickness)	469
LINK	252, 470

- LINK (create OLE control link to object from file)470
- LINK (specify CLASS link into project)252
- LIST350, 351, 352, 353, 354
- LIST (declare a window list control)350
- Local data154
- Local Derived Methods.....176
- Local menu290, 291
- Local tools.....295
- LOCALE.....759
- LOCALE (load environment file).....759
- Locate1004
- LOCK.....760
- LOCKTHREAD761
- LOCKTHREAD (re-lock the current execution thread)761
- Log1087
- LOG10762
- LOG10 (return base 10 logarithm)762
- Logarithm.....762, 763
- LOGE763
- LOGE (return natural logarithm).....763
- Logical Evaluations.....540
- Logical Expressions.....545
- Logical Operators540
- Logout.....1088
- LOGOUT.....764
- LOGOUT (begin transaction).....764
- LogoutIsolationLevel.....1089
- LOOP.....588, 589, 590
- LOOP (iteration structure)588
- LoseFocus999
- LOWER.....767
- LOWER (return lower case)767
- Maintaining INI Files727
- MAP45, 46
- MAP (declare PROCEDURE prototypes)..45
- MARK471
- MARK (set multiple selection mode)471
- MASK.....472
- control-level.....472
- MASK (set pattern editing data entry)472
- MATCH.....768, 770, 771
 - NoCase768
 - RegExp768
 - Simple768
 - Soundex768
- Wild768
- MATCH (return matching values)768
- MAX473
- MAX (set maximize control or total maximum).....473
- MaxHeight.....1040
- Maximize.....999
- MAXIMIZE.....475
- MAXIMIZE (set window open maximized)475
- Maximized.....999
- MAXIMUM.....772
- MAXIMUM (return maximum subscript value).....772
- MaxStatements.....1089
- MaxWidth.....1041
- MDI476
- MDI (set MDI child window)476
- MDI application window297
- MDI child windows297
- MDI frame window279
- MDI program297
- MEMBER43, 44
- MEMO192
- BINARY232
- MEMO (declare a text field)192
- Memory allocation
 - Dynamic153
 - Static153
- memory QUEUE220
- Memory redeclaration (OVER)263
- MENU355, 356
- MENUBAR290, 291, 292, 293
- MENUBAR (declare a pulldown menu) ...290
- MESSAGE773, 774
- MESSAGE (return message box response)773
- MessageHook.....1041
- Method Syntax Overview.....978
- Methods
 - VIRTUAL175, 176, 179, 181, 182, 183
- methods (PROCEDURES).....174
- MinHeight.....1042
- MinWidth.....1042
- Mixed data types.....171
- MM
 - REPORT attribute517

modal	286, 289, 478	NOBAR	480
MODAL (set system modal window)	478	NOBAR (set no highlight bar)	480
Modal Events	1008	NOCASE	258, 481
Modal Windows	476, 478	NOCASE (case insensitive KEY or INDEX)	258
Modeless Windows	476, 478	NOCASE (case insensitive report BREAK)	481
MODULE	47, 253	NOFRAME	423
MODULE (specify CLASS member source file)	253	NoHeight	1045
MODULE (specify MEMBER source file) ..	47	NOMEMO	782
Module data	154	NOMEMO (read file record without reading memo)	782
Module operator	538	NOMERGE	482
MONTH	776	NOMERGE (set merging behavior)	482
MONTH (return month of date)	776	Nonstop mode	854
Mouse Events	1005	Non-Trappable Run Time Errors	1103
MouseDown	1005	NOSHEET	484
MouseDown	1005	NOSHEET (set "floating" TABs)	484
MouseMove	1005	NOT	540
MouseOut	1005	Not implemented yet	1125
MouseUp	1005	NOTIFICATION	784
MOUSEX	777	NOTIFY	783
MOUSEY	778	NoTips	1046
Move	999	NoWidth	1045
Moved	999	Null	197
MSG	479	NULL	563, 728, 786, 871, 873, 874, 875
MSG (set status bar message)	479	Null Data Processing	197
Multiple Document Interface (MDI)	297	Null String	542
Multiplication operator	538	NUMERIC	788
Multi-threading		NUMERIC (return numeric string)	788
LOCKTHREAD	761	Numeric and Currency Pictures	159, 160
START	897	Numeric Constants	541
THREAD	911	Numeric Expressions	543
THREADLOCKED	912	Numeric Pictures	159
UNLOCKTHREAD	920	NumTabs	1047
NAME	779	Object declaration	
NAME (return file name)	779	CLASS 174, 175, 176, 177, 179, 180, 181, 182	
NAME (set prototype's external name)	74	Object Linking and Embedding	957
Name Mangling and C++ Compatibility	86	Object Properties (Encapsulation)	175
named group	149	Octal (numeric constant)	541
named GROUPs and QUEUEs	149	OCX Callback functions	973
Natural logarithm	763	OCX Events	975
NEW	156	OCX Propertys	973
NEW (allocate heap memory)	156	OCXGETPARAM	991
NewSelection	1006	OCXGETPARAM (return current event parameter string)	991
NEXT	780		
NextField	1043		
NextPageNo	1044		
NextTabStop	1044		

OCXGETPARAMCOUNT	990	OPEN (open a data structure)	791
OCXGETPARAMCOUNT (return number of parameters for current event)	990	OPEN (open OLE control object from file)	485
OCXLOADIMAGE	993	Open-mode activation.....	958
OCXLOADIMAGE (return an image object)	993	OpenWindow	1000
OCXREGISTEREVENTPROC	986	Operating Assignment Statements.....	560
OCXREGISTEREVENTPROC (install event processing callback)	986	Operating Assignments	560
OCXREGISTERPROPCHANGE.....	985	Operator Precedence	537
OCXREGISTERPROPCHANGE (install property change callback).....	985	Operators	
OCXREGISTERPROPEDIT	984	Conditional Operators	540
OCXREGISTERPROPEDIT (install property edit callback)	984	Logical Operators.....	540
OCXSETPARAM	992	OPT.....	260
OCXSETPARAM (set current event parameter string).....	992	OPT (exclude null KEY or INDEX entries)	260
OCXUNREGISTEREVENTPROC.....	989	OPTION	361, 362, 363
OCXUNREGISTEREVENTPROC (un-install event process callback)	989	OPTION (declare a set of RADIO controls)	361
OCXUNREGISTERPROPCHANGE	988	OR.....	540
OCXUNREGISTERPROPCHANGE (un- install prop change callback).....	988	OROF.....	582, 583
OCXUNREGISTERPROPEDIT	987	Out of attribute space	1124
OCXUNREGISTERPROPEDIT (un-install property edit callback).....	987	outline control	433
OEM.....	259	OVER.....	263
OEM (set international string support)....	259	OVER (set shared memory location).....	263
OF.....	582, 583	Overflow	
Old symbol non-NIL.....	1125	arithmetic.....	159, 566
OLE.....	357, 358, 359, 360, 957, 958	Overview	537
container windows	957	OVR	461
OLE Automation	957	OWNER	264
OLE Controller application.....	957, 958	OWNER (declare password for data encryption).....	264
OLE custom control OverView	970	PACK	795
OLE Overview	957	PACK (remove deleted records).....	795
OLE Server application.....	957, 958	Packed Decimal.....	123, 124, 125
OLEDIRECTORY	968	PAGE	486
OLEDIRECTORY (get list of installed OLE/OCX)	968	PAGE (set page total reset).....	486
OMIT.....	96, 97	Page Overflow	318
OMIT (specify source not to be compiled).96		PAGEAFTER	487
OMITTED.....	789, 790	PAGEAFTER (set page break after)	487
ONCE	98	Page-based printing.....	306
ONCE (prevent duplicate included data)...98		PAGEBEFORE	488
OPEN.....	485, 791, 792, 793, 794	PAGEBEFORE (set page break first).....	488
		PageDown	1006
		PAGENO.....	489
		PAGENO (set page number print).....	489
		PageUp	1006
		PALETTE	490

PALETTE (set number of hardware colors)	Polymorphism	174
.....	POPBIND	555
PANEL	POPBIND (restore runtime expression string	
364	name space)	555
PAPER.....	POPERRORS	820
491	POPUP	806
PAPER (set report paper size)	POPUP (return popup menu selection) ..	806
491	POSITION.....	808, 810
Parameter Passing to OLE/OCX Methods	POSITION (return record sequence position)	
.....	808
981	POST	811
Parameters	PRAGMA	812
544	PRE.....	265
expression used as	PreAlertKey.....	1000, 1006
537, 549	PRESS.....	814
Parent	PRESS (put characters in the buffer)	814
1048	PRESSKEY.....	815
PARENT	PRESSKEY (put a keystroke in the buffer)	
175, 179, 181	815
PASSWORD.....	PREVIOUS	816
492	PREVIOUS (read previous view record in	
PASSWORD (set data non-display)	sequence).....	816
492	PrevTabStop.....	1044
PATH	PRIMARY.....	267
796	PRIMARY (set relational primary key)	267
PATH (return current directory)	primary key	267
796	PRINT	818
Pattern Pictures	PRINT (print a report structure)	818
167	Print structure	
PDECIMAL	BREAK	307
124	DETAIL.....	308, 309
PDECIMAL (signed packed decimal)	FOOTER	310, 311
124	FORM.....	312
PEEK	HEADER	313, 314
797	PRINTER	819
PEEK (read memory address).....	built-in variable	546
797	Printer Control Properties	315
PENCOLOR	PRINTERDIALOG	819
798	PRINTERDIALOG (return chosen printer)	
PENCOLOR (return line draw color)	819
798	PrinterDialogHook.....	1049
PENSTYLE	PrintMode.....	1049
799	PRIVATE.....	268
PENSTYLE (return line draw style)	PRIVATE (set procedure private to a CLASS	
799	or module)	75
PENWIDTH.....	PRIVATE (set variable private to a CLASS	
800	module)	268
PENWIDTH (return line draw thickness) .		
800		
Picture Tokens.....		
159		
Pictures		
Date.....		164, 165
Key-in.....		168
Numeric and Currency		159, 160
Pattern.....		167
Scientific Notation		162
String.....		163
Time		166
PIE		801, 802
Pixels		1048
POINTER.....		803, 1131
POINTER (return last queue entry position)		
.....		803
POINTER (return relative record position)		
.....		1131
POINTS		
REPORT attribute		517
POKE		804
POKE (write to memory address).....		804
POLYGON		805

PROC (set function called as procedure without warnings)	76	CHECK.....	407
PROCEDURE.....	48, 49, 50	Checked	1014
PROCEDURE (define a procedure)	48	Child	1015
Procedure Overloading.....	83	ChildIndex	1015
PROCEDURE Return Types	69	ChoiceFeq.....	1016
Profile.....	1089	ClientHandle.....	1016
PROGRAM	41, 42	ClientWndProc	1017
PROGRAM (declare a program)	41	CLIP	408, 959
Progress	1050	ClipBits	1018
PROGRESS	367, 368	COLOR.....	410, 411
PROJECT	216	ColorDialogHook	1019
PROJECT (set view fields)	216	COLUMN.....	412
PROMPT	365, 366	COMPATIBILITY	414, 959
Prop	413	Completed	1081
EXTEND.....	413	ConnectString	1081
PROP.....	1010	Create.....	200, 204
ABOVE.....	466	CREATE	234, 415, 959, 962, 963, 971
ABSOLUTE	391	Ctrl.....	971
AcceptAll	1009	CurrentKey	1082
ACTIVE	1010	CURSOR.....	416
ALONE	392	DDEMode.....	1019
ALRT	393	DDETimeOut.....	1020
AlwaysDrop.....	1010	Deactivate	963
ANGLE	395	DECIMAL	467
AppInstance	1011	DecimalOffset.....	467
Ascending	202	DEFAULT	417
AssertHook	1011	DeferMove.....	1020
AssertHook2	1012	DELAY.....	418
AT.....	396, 397, 398	DesignMode	971
AUTO	399	Details	1083
AutoPaper	1013	DISABLE	419
AUTOSIZE	400, 959, 963	DOCK	420
Background.....	410, 411	DOCKED	421
BELOW	466	DOCUMENT.....	422
BEVEL.....	402	DOUBLE.....	423
BINARY.....	202, 232	DoVerb	960, 971
Blob	959, 962, 964	DRAGID	424
Blobs	198, 204	Driver	200
BOXED.....	403	DRIVER.....	239
BreakVar	1013	DriverLogoutAlias	1083
Buffer.....	1014	DriverString	200
ButtonFeq	526	DROP	425
CAP	404	DROPID	426
CENTER	405, 467, 468	DropWidth	425
CENTERED	406	DUP	201, 204, 240
CenterOffset.....	467	Edit	1022
		Enabled	1024

Encrypt.....	200	ImageBits	1035
ENCRYPT	241	ImageBlob	1036
EventsWaiting	1026	IMM	459, 460
ExeVersion	1026	InitAStringHook	1036
FalseValue	529	INNER	251
FatalErrorHook.....	1026	INS	461
Feq	526	Interface	1036
FetchSize	632, 1083	InToolbar	1036
Field	1084	IsRadio	971
FIELD.....	198, 199, 200, 202, 203	Items.....	1037
Fields.....	198, 201, 202	JOIN	462
FieldsFile.....	1084	JoinExpression.....	217
File	1085	Key	198, 199, 200, 201, 202, 463
FileDialogHook.....	1026	Keys	198
Files.....	1085	Label....	198, 199, 200, 201, 202, 204, 208
FILL	427	LANDSCAPE	464
FILTER.....	245, 247	Language	961, 972
FIRST	428	LAST	428
FLAT	429	LastEventName.....	971
FlushPageNumFunc	1027	LazyDisplay	1037
FlushPreview	1028	LEFT.....	466, 467, 468
Follows	1029	LeftOffset.....	467
FONT	430, 431, 432	LFNSupport.....	1038
FontCharSet.....	430	LibHook	1038
FontColor	430	LibVersion	1039
FileDialogHook	1029	Line.....	1039
FontName	430	LineCount.....	1039
FontSize	430	LineHeight	1040
FontStyle	430	LINEWIDTH.....	469
FORMAT	433	LINK	470
FROM.....	448	ListFeq.....	526
FULL	450	Log	1087
GlobalHelp	1086	Logout	1088
GrabHandles	971	LogoutIsolationLevel	1089
GRAY	451	MARK	471
HaltHook	1030	MASK	472
Handle.....	1030	Max.....	292, 293, 473
HeaderHeight	1031	MaxHeight	1040
Height.....	396	MAXIMIZE	475
Held.....	1086	MaxStatements	1089
HIDE.....	453	MaxWidth	1041
HLP	454	MDI.....	476
HSCROLL	455	Memos.....	198
HscrollPos	1032	MessageHook	1041
HVSCROLL.....	455	MinHeight	1042
ICON	456	MinWidth	1042
IconList.....	1033	MM	517

MODAL	478	PrinterDialogHook	1049
MSG	479	PrintMode	1049
Name.....	198, 199	Profile	1089
NAME	254, 255	Progress	1050
NextField	1043	ProgressEvents	1090
NextPageNo.....	1044	RANGE.....	495
NextTabStop	1044	RangeHigh	495
NOBAR	480	RangeLow	495
NoCase	201, 204	READONLY.....	496
NOCASE	258	Reclaim	200
NOFRAME	423	RECLAIM	270
NoHeight	1045	Record.....	1093
NOMERGE	482, 483	RejectCode.....	1052
NOSHEET	484	REPEAT	497
NoTips.....	1046	ReportException.....	971
NoWidth	1045	REQ.....	498
NumTabs.....	1047	RESET	499
Object. 959, 960, 961, 962, 963, 964, 966, 967		RESIZE	423, 500
OEM.....	259	RIGHT	467, 501
OLE 959, 960, 961, 962, 963, 964, 965, 966, 967, 971, 972		RightOffset	467
OPEN	485, 959, 960, 963	ROUND	502
OPT.....	201, 204, 260	SaveAs.....	971
ORDER	261	ScreenText.....	1053
Over	203	SCROLL.....	504
OVR	461	Selected	1054
Owner.....	200	SelEnd.....	1054
OWNER	264	SelStart.....	1054
PAGE	486	SINGLE	506
PAGEAFTER	487	Size	202, 1055
PageAfterNum	487	SKIP	507
PAGEBEFORE	488	SnapHeight	1057
PageBeforeNum	488	SnapWidth.....	1056
PAGENO	489	SPREAD.....	508
PALETTE	490	SQLDriver.....	1093
Parent.....	1048	SQLJoinExpression.....	217
PASSWORD	492	STATUS	509
Paste	960, 963, 964	StatusFont	1058
PasteLink	964	StatusText	509
Pixels.....	1048	STD	511
Places	202	STEP	512
POINTS.....	517	StmtAttr	1093
PREVIEW.....	493, 494	StopHook.....	1058
PrevTabStop	1044	STRETCH	513, 959
Primary.....	201, 204	SupportsOp	1094
PRIMARY	267	SupportsType	1094
		SYSTEM.....	515
		TabRows	1059

TALLY	516	Ypos	396
Target.....	1060	Zoom	959
TempImage.....	1060	PROP:DriverString.....	239
TempImagePath	1060	PROP:PropVscroll	1051
TempImageStatus.....	1060	PROP:SMOOTH.....	508
TempNameFunc	1060	PROP:Vertical.....	531
TempPagePath	1062	Properties	
TempPath.....	1062	Object. 174, 175, 176, 177, 178, 179, 180,	
Text.....	389, 1094	181	
THOUS.....	517	Property Access Syntax.....	546
Thread.....	200, 204, 1062	Property Equates	389
THREAD	272	Property Expressions.....	546
Threading	1062	PROPLIST	433
TILED	518	BackColor.....	443
TIMER.....	519	BackSelected	443
TIP.....	520	CellStyle	433
TipDelay	1063	Center.....	433
TipDisplay	1063	CenterOffset.....	433
TipsFont	1063	Color	434, 437
TOOLBOX.....	522	ColStyle	433
Touched	1064	Decimal	433
TRN.....	524	DecimalOffset.....	433
TrueValue	529	DefaultTip	433
Type	200, 208, 1066	Exists.....	443
Update.....	960	FieldNo	433
UPR.....	404	Fixed.....	435
USE.....	526	Format.....	433, 434, 435, 437
UseAddress	1068	GRID	452
Value	1095	Group	434, 435, 436
VCR.....	530	Header.....	435
VcrFeq.....	530	HeaderCenter.....	433
Visible.....	1069	HeaderCenterOffset	433
VLBproc	1071	HeaderDecimal	433
VLBval.....	1071	HeaderDecimalOffset.....	433
VScroll.....	1051	HeaderLeft.....	433
VscrollPos	1075	HeaderLeftOffset.....	433
WALLPAPER	532	HeaderRight	433
Watched	1077	HeaderRightOffset	433
WheelScroll.....	1077	Icon.....	434, 437
Width.....	396	IconTrn	433
WindowsVersion	1077	LastOnLine	433
WITHNEXT	533	Left	433, 434
WITHPRIOR	534	LeftOffset.....	433
WIZARD	535	Locator	435
WndProc	1078	MouseDownField	445
XOrigin	1080	MouseDownRow	445
Xpos.....	396	MouseDownZone	445

MouseMoveField.....	445	PROPSTYLE	441
MouseMoveRow	445	BackColor.....	441
MouseMoveZone	445	BackSelected	441
MouseUpField.....	445	CharSet	441
MouseUpRow	445	FontColor.....	441
MouseUpZone	445	FontName.....	441
Picture	433	FontSize	441
Resize	433	FontStyle	441
Right.....	433, 435	Picture	441
RightBorder	433	TextColor.....	441
RightOffset	433	TextSelected	441
Scroll	435	PROTECTED.....	269
TextColor.....	443	PROTECTED (set procedure private to a	
TextSelected	443	CLASS or derived CLASS)	77
Tip	436	PROTECTED (set variable private to a	
Tree.....	434, 437	CLASS or derived CLASS)	269
TreeBoxes.....	433	Prototype Syntax	58
TreeIndent.....	433	PSTRING	133
TreeLines	433	PSTRING (embedded length-byte string)	133
TreeOffset	433	PUSHBIND	556
TreeRoot	433	PUSHBIND (save runtime expression string	
Underline.....	433	name space)	556
Width.....	433, 435, 436	PUSHERRORS	822
PROPPRINT.....	315	PUT	825, 826
Collate	315	PUT (re-write record)	824
Color.....	315	PUTINI	828
Context.....	315	PUTINI (set INI file entry).....	828
Copies	315, 318	PUTREG	829
Device	315	QUEUE	220, 221, 222
DevMode.....	315	ADD	612, 613, 614
Driver.....	317	CHANGES.....	647
Duplex.....	316	POINTER	803
FontMode	315	QUEUE (declare a memory QUEUE	
FromMin	315	structure)	220
FromPage	315	QUOTE	830
Paper.....	317, 491	RADIO.....	369, 370
PaperBin	315	RANDOM	831
PaperHeight	315, 491	RANDOM (return random number)	831
PaperWidth	315, 491	RANGE	495
Percent.....	317	RANGE (set range limits)	495
Port.....	317	Range validation	741
PrintToFile.....	315	RangeHigh	495
PrintToName	315	RangeLow.....	495
Resolution	317, 318	RAW (pass address only)	78
ToMax	315	READONLY	496
ToPage.....	315	READONLY (set display-only).....	496
Yresolution	315	REAL.....	116

REAL (eight-byte signed floating point) ...	116	RESET (reset record sequence position)	842
RECLAIM.....	270	RESET (set total reset).....	499
RECLAIM (reuse deleted record space) .	270	RESIZE	423, 460, 500
Record.....	1093	RESIZE (set variable height TEXT control)	
RECORD	196	500
RECORD (declare record structure)	196	Restore	1000
RECORDS.....	832, 833	Restored	1000
RECORDS (return number of rows in data		RESTORESTATE	845
set)	832	RESUME.....	846
Recursive		RETURN.....	597
FUNCTION	154	RETURN (return to caller)	597
PROCEDURE	154	return file existence.....	702
Redeclares (OVER).....	263	Rewrite (PUT)	824
Reference Assignment Statements.....	563	RIGHT.....	466, 467, 468, 501, 847
Reference Assignments	563	RIGHT (return right justified string).....	847
Reference Item Format.....	27	RIGHT (set MENU position)	501
Reference Variables	150	ROLLBACK.....	848
REGET	837	ROLLBACK (terminate unsuccessful	
REGION.....	372, 373	transaction).....	848
REGISTER	834	ROUND.....	502, 849
REGISTER (register event handler)	834	ROUND (return rounded number)	849
regular expression match	768	ROUND (set round-cornered BOX)	502
RejectCode	1052	ROUTINE.....	53, 54
REJECTCODE	836	DO	594
REJECTCODE (return reject code number)		EXIT	595
.....	836	RTF	503
Rejected.....	1006	RTF Support	503
RELEASE	839	Rules for Procedure Overloading	84
Remainder (Modulus division)	538	RUN	851
REMOVE	840	RUN (execute command)	851
REMOVE (erase a file)	840	Run Time Errors	1097, 1103
RENAME	841	RUNCODE.....	853
RENAME (change file directory name) ...	841	RUNCODE (return program exit code) ...	853
REPEAT	497	Runtime Expression.....	549
REPEAT (set repeat button rate)	497	Runtime Expression Evaluation.....	549
Repeat count notation	542	Runtime Property Assignment	
Repeated characters	620	Property Access Syntax	546
REPLACE (set replacement constructor or		Property Equates	389
destructor)	79	Scientific Notation Pictures.....	162
REPORT.....	304, 305, 306	scope	153
REPORT (declare a report structure).....	303	Screen Fields (controls).....	298
Report totals		ScreenText.....	1053
MAX	473	SCROLL.....	504
REQ	498	SCROLL (set scrolling control)	504
REQ (set required entry)	498	ScrollBottom.....	1006
Reserved Words.....	38	ScrollDown.....	1006
RESET	499, 842, 844	ScrollDrag	1007

- ScrollTop..... 1007
 ScrollTrack..... 1007
 ScrollUp..... 1007
 SECTION..... 99
 SECTION (specify source code section) ... 99
 SELECT..... 854, 855
 SELECT (select next control to process) 854
 Selected..... 1007, 1054
 SELECTED..... 856
 SELECTED (return control that has received focus) 856
 SelEnd..... 1054
 SELF..... 180, 181, 183
 SelStart..... 1054
 SEND..... 857
 SEND (send message to file driver) 857
 SEPARATOR..... 505
 SEPARATOR (set separator line ITEM) .. 505
 SET..... 858, 859
 SET3DLOOK..... 862
 SETCLIPBOARD..... 863
 SETCLIPBOARD (set windows clipboard contents) 863
 SETCLOCK..... 864
 SETCLOCK (set system time)..... 864
 SETCOMMAND..... 865
 SETCOMMAND (set command line parameters)..... 865
 SETCURSOR..... 866
 SETFONT..... 868
 SETKEYCHAR..... 869
 SETKEYCHAR (specify ASCII code) 869
 SETKEYCODE..... 870
 SETKEYCODE (specify keycode)..... 870
 SETNULL..... 873
 SETNULL (set file field null) 873
 SETPATH..... 876
 SETPENCOLOR..... 877
 SETPENSTYLE..... 878
 SETPENWIDTH..... 879
 SETPOSITION..... 880
 SETTARGET..... 546, 547, 881
 SHARE..... 1132
 SHEET..... 374, 375, 376
 Short to Long filename conversion
 LONGPATH..... 766
 SHORTPATH..... 884
 SHUTDOWN..... 886
 Simple Assignment Data Type Conversion..... 571
 Simple Assignment Statements..... 559
 Simple Assignments..... 559
 SINGLE..... 506
 SINGLE (set TEXT for single line entry) .. 506
 Single Document Interface (SDI)..... 297
 Single-precision real..... 114, 118
 Size..... 1055
 SIZE..... 100
 SIZE (memory size in bytes)..... 100
 Size Window..... 1000
 Sized..... 1000
 SKIP..... 507, 888, 889
 SKIP (set Tab key skip or conditional print control) 507
 SMOOTH..... 508
 SnapHeight..... 1056
 SnapWidth..... 1057
 SORT..... 890
 SORT (QUEUE)..... 890
 SORT (sort queue entries)..... 890
 sort order
 INDEX..... 189
 KEY..... 190, 191
 VIEW..... 261
 Sound (BEEP)..... 627
 soundex..... 768
 Source variable..... 559
 Special Characters..... 39
 used in string constants..... 542
 Specific Errors..... 1107
 SPIN..... 378, 379
 SPIN (declare a spinning list control) 377
 SPREAD..... 508
 SPREAD (set evenly spaced TAB controls) 508
 SQLCALLBACK (register or unregister a SQLCallBackInterface)..... 893
 SQLDriver..... 1093
 SQRT..... 896
 SQRT (return square root)..... 896
 SREAL..... 114
 SREAL (four-byte signed floating point) .. 114
 Standard Date..... 29
 Standard Time..... 30

START	897, 898	System Date	
START (return new execution thread).....	897	TODAY	915
Statement Execution Sequence	56	System menu	515
Statement Format.....	33	System modal	286, 478
STATIC.....	271	System Time	
STATIC (set allocate static memory).....	271	CLOCK	656
Static Data	153	SETCLOCK.....	864
STATUS.....	509, 510, 899	TAB.....	383, 384
STATUS (set status bar)	509	TabChanging	1007
StatusFont	1058	TabRows.....	1059
STD.....	511	TALLY	516
STD (set standard behavior)	511	TALLY (set total calculation times)	516
STEP	512	TAN.....	910
STEP (set SPIN increment).....	512	TAN (tangent of angle)	910
StmtAttr.....	1093	Target.....	1060
STOP	901	TARGET	
STOP (suspend program execution).....	901	built-in variable	546
StopHook	1058	Templmage.....	1060
STREAM.....	902	TemplmagePath	1060
STREAM (enable operating system		TemplmageStatus	1060
buffering).....	902	TempNameFunc	1060
STRETCH.....	513	TempPagePath.....	1062
STRETCH (set OLE object stretching)....	513	TempPath	1062
STRING	126, 380, 381, 382	Termination	
STRING (declare a string control)	380	FUNCTION.....	597
STRING (fixed-length string)	126	HALT	732
String Constants	542	PROCEDURE	597
String Expressions.....	544	PROGRAM.....	597
String not CCST	1126	ROUTINE	595
String Pictures	163	Text.....	1094
String Slicing.....	136	TEXT.....	385, 386, 387
STRPOS.....	903	The Concatenation Operator	539
Structure Termination	35	THEN	586
SUB	907, 908	THOUS	
SUB (return substring of string).....	907	MM	
sub-classing.....	995	POINTS (set report coordinate	
Subscript		measure).....	517
Array.....	235, 236	REPORT attribute	517
MAXIMUM.....	772	Thread.....	1062
Subtraction operator	538	THREAD	272, 273, 911
SupportsOp	1094	THREAD (return current execution thread)	
SupportsType	1094	911
SUSPEND	909	THREAD (set thread-specific memory	
Switch To.....	515	allocation).....	272
SYSTEM.....	515	threading.....	846
built-in variable.....	546	Threading.....	1062
SYSTEM (set system menu)	515	THREADLOCKED	912

- THREADLOCKED (returns current execution thread locked state).....912
 TIE(associate a string value to an ASTRING).....913
 TIED(retrieves a value associated with an ASTRING914
 TILED
 IMAGE attribute518
 TILED (set tiled image).....518
 Time
 Standard Time30
 TIME139
 TIME (four-byte time).....139
 Time Pictures.....166
 Timer.....1001
 TIMER.....519
 TIMER (set periodic event).....519
 TIMES588, 589
 TIP520
 TIP (set "balloon help" text)520
 TipDelay.....1063
 TipDisplay.....1063
 TipsFont.....1063
 TO.....582, 583, 588, 589, 590
 TODAY915
 TODAY (return system date)915
 TOGETHER.....521
 TOOLBAR.....294, 295, 296
 TOOLBAR (declare a tool bar)294
 TOOLBOX522, 523
 TOOLBOX (set toolbox window behavior)522
 Totals
 AVE401
 CNT409
 MAX473, 474
 MIN.....477
 SUM514
 Touched.....1064
 Transaction Processing
 COMMIT.....663
 LOGOUT764
 ROLLBACK.....848
 Trappable Run Time Errors1097
 Tree control434
 tree controls in list boxes350
 TRN524
 TRN (set transparent control)524
 Truncation
 Data Type Conversion Rules566
 two-column drop menu291
 TYPE.....274, 916
 TYPE (specify PROCEDURE type definition)
 TYPE (type definition).....274
 TYPE (write string to screen).....916
 Type Conversion.....566, 570
 Type Conversion and Intermediate Results570
 type definition.....274
 Type descriptor not static.....1124
 UNBIND557, 558
 UNBIND (free runtime expression string variable).....557
 UNHIDE917
 UNHIDE (show hidden control).....917
 Unknown assignment operator1124
 Unknown case type1124
 Unknown clear type1124
 Unknown descriptor type1124
 Unknown designator kind1124
 Unknown equate type1124
 Unknown errors1124
 Unknown expression kind.....1124
 Unknown expression type.....1124
 Unknown formal entity1124
 Unknown initializer type1124
 Unknown label/routine1124
 Unknown operator1124
 Unknown parameter kind.....1124
 Unknown picture type1124
 Unknown screen structure kind1124
 Unknown simple formal type.....1124
 Unknown special identifier1124
 Unknown static label.....1124
 Unknown string kind1124
 Unknown structure field1124
 Unknown variable context.....1124
 Unknown variable type1124
 UNLOAD918
 UNLOAD (remove a CALLED DLL from memory)918
 UNLOCK919
 UNLOCK (unlock a locked data file)919
 UNLOCKTHREAD920

UNLOCKTHREAD (unlock the current execution thread)	920	SIGNED.....	112
UNQUOTE.....	921	SREAL.....	115
UNQUOTE language statement.....	921	STRING.....	126, 127
UNREGISTER.....	922	TIME.....	139, 140
UNREGISTER (unregister event handler)	922	ULONG.....	111
UNSIGNED.....	113	UNSIGNED	113
UNSIGNED (16/32-bit unsigned integer)	113	USHORT	107
UNTIE(disassociate a string value from an ASTRING).....	923	Variable-size arrays	155
UNTIL	588, 589, 590	VARIANT	141
UP.....	525	VB to Clarion Syntax.....	978
UPDATE.....	924	VCR	530
UPDATE (write from screen to USE variables).....	924	VCR (set VCR control).....	530
UPPER.....	925	VERTICAL	531
UPPER (return upper case).....	925	VIEW.....	212, 213, 215
UPR.....	404	VIEW (declare a "virtual" file).....	212
USE.....	526, 527	VIRTUAL (set virtual method).....	82
USE (set field equate label or control update variable)	526	VIRTUAL Methods.....	175
UseAddress	1068	Visible	1069
VAL.....	926	VLBproc	1071, 1072
VAL (return ASCII value).....	926	VLBval.....	1071
valid Clarion label	34	VSCROLL	455
Value.....	1095	VscrollPos	1075
VALUE.....	529	WALLPAPER.....	532
Value not static	1124	WALLPAPER (set background image)....	532
Variable Size Declarations	155	WATCH.....	927
Variables		WATCH (automatic concurrency check) ..	927
ANY.....	141, 142	Watched.....	1077
BFLOAT4	119	WHAT	928
BFLOAT8	121	WHAT (return field from group)	928
BYTE.....	103	WheelScroll.....	1077
CSTRING.....	130, 131	WHERE.....	929
DATE.....	137, 138	WHILE.....	588, 589, 590
DECIMAL	122, 123	wild card match.....	768
GROUP.....	171, 172, 173	WINDOW	283, 284, 285, 286, 287, 288
Implicit.....	147, 148	MDI child window	286, 288
LONG.....	108, 109	non-MDI window	283
PDECIMAL.....	125	WINDOW (declare a dialog window)	283
PRIVATE.....	268	Window Controls and Input Focus.....	298
PROTECTED.....	269	Window Functions	609
PSTRING	134	Window Overview	297
REAL.....	116, 117	Windows Standard Dialog Functions.....	661
SHORT.....	105	WindowsVersion	1077
		WITHNEXT	533
		WITHNEXT (set widow elimination)	533
		WITHPRIOR	534
		WITHPRIOR (set orphan elimination)	534
		WIZARD.....	535

WIZARD (set "tableless" SHEET control) ..535	YIELD.....932, 933
WndProc..... 1078	YIELD (allow event processing)932
XOR540	ZOOM536
YEAR931	ZOOM (set OLE object zooming)536
YEAR (return year of date).....931	

