

Database Drivers



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 (0903)

Contents:

Database Drivers.....	5
Overview - Data Independence	5
Choosing the Right Database Driver.....	6
Common Database Driver Features.....	7
Driver Strings.....	10
ISAM File Drivers.....	11
ASCII File Driver	11
Basic Database Driver	21
Btrieve Database Driver.....	31
Clarion Database Driver.....	49
Clipper Database Driver.....	59
dBaseIII Database Driver.....	75
dBaseIV Database Driver	91
DOS Database Driver	109
FoxPro / FoxBase Database Driver	117
TopSpeed Database Driver	133
All SQL Accelerators (Drivers)	153
General Information for all SQL Drivers.....	153
Using SQL Tables in your Clarion Application.....	154
SQL Driver Behavior	157
Performance Considerations	159
Date and Time Column Considerations.....	161
SQL Batch Transaction Processing.....	162
Using Embedded SQL	163
Runtime SQL Properties for Views using SQL Drivers.....	169
VIEW support for aggregate functions.....	171
Debugging Your SQL Application	173
SQL Accelerator Drivers:Supported Commands and Attributes.....	175
CHECKFORNULL.....	180
SQL Driver Strings(Generic).....	181
SQL Driver Properties(Generic).....	195
ADO Database Driver	208
MSSQL Accelerator	212
ODBC Accelerator Driver.....	238
Oracle Accelerator	256
Scalable SQL Accelerator Driver	285
SQLAnywhere Accelerator.....	295
Index:.....	311

Database Drivers

Overview - Data Independence

Clarion achieves database independence with its built-in driver technology, which lets you access data from virtually any file system using the same set of Clarion language commands. Many file drivers are available and more are being added.

Tip

Before you can use a database driver, it must be registered. The database drivers in this package are pre-registered. See the *User's Guide--Configuring the Environment* for more information on registering add-on database drivers.

The Clarion language commands for accessing data are the same regardless of the file system you use; simply choose the appropriate file driver from the drop-down list in your Data Dictionary, then don't worry about it. The file driver translates the Clarion commands to the chosen file system's native format. The drivers read and write in the file system's native format without temporary files or import/export routines.

Choosing the Right Database Driver

Choosing a file system is an important decision, and we encourage you to gather as much information from as many good sources as you can to support your decision. Although the choice of file systems is important, with Clarion, it is not irrevocable. If the file system you choose does not live up to your expectations, you can change to one that does. For example, some developers use the TopSpeed file system for project development, then switch to an SQL file system during project implementation in order to postpone the expense of the SQL software and server hardware until late in the development cycle.

Common Database Driver Features

Importing File Definitions

For existing data, you can generally import file definitions into your Clarion data dictionary. We strongly recommend importing file definitions whenever possible, because it reduces your development time and effort, plus it results in fewer errors in file definitions. See *The Dictionary Editor* in the *User's Guide* for more information on importing files.

Keys, Indexes, and Performance

Although you may define indexes within your Clarion data dictionary that do not exist within the native file system, we do not recommend doing so because your application performance will generally suffer. Instead, we recommend defining the required key or index with the native file system's tools, then importing the file definition, including the key or index definitions, into your Clarion data dictionary.

Sorting and Collating Sequences

By default, all SoftVelocity's database drivers sort using the ANSI collating sequence. Adding the OEM attribute causes the driver to use the ASCII collating sequence.

Disk Caching and Data Integrity

Disk caching can interfere with the data integrity features of many file systems. By disk caching, we mean any facility (for example SMARTDRV) that tells the database driver that a record was written to the disk when in fact it was not.

To improve performance, disk-caching facilities typically accumulate several records at a time in RAM then write them to disk all at once. While this does improve performance, it can result in corrupt data files if the system fails (due to a power outage, etc.) before the records are written to disk.

A reliable Uninterruptible Power Supply (UPS) can drastically reduce this risk. Therefore, we generally recommend no disk caching, but if you must cache, then be sure to use a reliable UPS.

Database Driver System-wide Logging

All of SoftVelocity's database drivers can create a log file documenting Clarion I/O statements they process, and the SQL Accelerator drivers can log the corresponding SQL statements, and the SQL return codes.

You can generate system-wide logs and on-demand logs (conditional logging based on your program logic).

A utility/example application is included--Trace.EXE. A compiled version is installed in the `..\bin` directory and the source `.APP` is installed in the `\Examples\Resource\Trace` directory. This utility allows you to easily set tracing options for each file driver and for the VIEW engine. These settings are stored in `WIN.INI`.

For system-wide logging, you can add the following to your `WIN.INI` file:

```
[filedriver]
Profile=[1|0]
Details=[1|0]
Trace=[1|0]
TraceFile=[Pathname]
```

where *filedriver* is the database driver name (for example `[MS-SQL]`). Neither the INI section name `[filedriver]` nor the INI entry names are case sensitive.

`Profile=1` tells the driver to include the Clarion I/O statements in the log file; `Profile=0` tells the driver to omit Clarion I/O statements. The Profile switch must be turned on for the Details switch to have any effect.

`Details=1` tells the driver to include record buffer contents in the log file; however, if the file is encrypted, you must turn on both the Details switch and the `ALLOWDETAILS` switch to log record buffer contents (see `ALLOWDETAILS`). `Details=0` tells the driver to omit record buffer contents. The Profile switch must be turned on for the Details switch to have any effect.

`Trace=1` tells the driver to include all calls to the back-end DBMS, including the generated SQL statements and their return codes, in the log file. `Trace=0` omits these calls. The Trace switch generally generates log information that helps to debug the SQL drivers, but is normally not particularly useful to the developer.

`TraceFile` names the log file to write to. If `TraceFile` is omitted the driver writes the log to *driver.log* in the current directory. *Pathname* is the full pathname or the filename of the log file to write. If no path is specified, the driver writes the specified file to the current directory.

Logging opens the named log file for exclusive access. If the file exists, the new log data is appended to the file.

On Demand Logging

For on-demand logging you can use property syntax within your program to conditionally turn various levels of logging on and off. The logging is effective for the target table and any view for which the target table is the primary table.

```
file{PROP:Profile}=Pathname      !Turns Clarion I/O logging on
file{PROP:Profile}=''           !Turns Clarion I/O logging off
PathName = file{PROP:Profile}    !Queries the name of the log file
file{PROP:Log}=string           !Writes the string to the log file
file{PROP:Details}=1           !Turns Record Buffer logging on
fFile{PROP:Details}=0          !Turns Record Buffer logging off
```

where *Pathname* is the full pathname or the filename of the log file to create. If you do not specify a path, the driver writes the log file to the current directory.

You can also accomplish on demand logging with a SEND() command and the LOGFILE driver string.

Driver Strings

There are switches or "driver strings" you can set to control the way your application creates, reads, and writes files with a specific driver. Driver strings are simply messages or parameters that are sent to the file driver at run-time to control its behavior. The various driver specific driver strings are described in the *Driver Strings* section for each driver.

Driver strings are sent in three ways: with the OPEN or CREATE statement, with the SEND procedure, and with property syntax.

DRIVER('Driver', '/DriverString = value')

The OPEN(file) and CREATE(file) statements send any driver strings specified in the FILE's DRIVER attribute. OPEN sends the string immediately *before* the file is opened. You may specify these driver strings with a hand coded FILE declaration (see *DRIVER* in the *Language Reference* for more information) or in the Data Dictionary (**Driver Options** field in the **File Properties** dialog-- see *File Properties*). In either case, you must prepend a forward slash (/) to the driver string. For example:

```
MyFile FILE,DRIVER('TopSpeed','/LOGFILE=MyFile.Log')
CODE
OPEN(MyFile)           !sends the LOGFILE driver string
```

SEND(file, 'DriverString')

The SEND function sends a driver string to the file driver at any time, including before the file is opened. SEND functions take two forms--*with* an equal sign to change the value of the switch, and *without* an equal sign to return the value of the switch. With SEND, the ISAM drivers do not require the preceding forward slash, but the SQL drivers do require it. For example:

```
SEND(MyFile,'LOGFILE='&MyLogFile)           !Set the logfile
MyLogFile=SEND(MySQLFile,'/LOGFILE')        !Query the logfile
OldLogFile=SEND(MyFile,'LOGFILE='&NewLogFile) !Set & Query the logfile
```

file{PROP:DriverString}

Property syntax is an alternative to the SEND function. With property syntax you can send a driver string to the file driver any time *after* the file is opened. With property syntax, the driver string does not require the preceding forward slash. For example:

```
MyLogFile = 'MyFile.Log'
MyFile{PROP:Profile}=MyLogFile !Set the logfile
MyLogFile = MyFile{PROP:Profile} !Query the logfile
```

ISAM File Drivers

ASCII File Driver

ASCII:Specifications

The ASCII driver reads and writes standard ASCII files without field delimiters. This is often used for mainframe data import/export with an ASCII flat-file. By default, a carriage-return/line-feed delimits records. The ASCII driver does not support keys.

Files:	C60ASCXL.LIB	Windows Static Link Library
	C60ASCX.LIB	Windows Export Library
	C60ASCX.DLL	Windows Dynamic Link Library

Tip

Due to its lack of relational features and security (anyone can view and change an ASCII file using Notepad), it's unlikely you'll use the ASCII driver to store large data files. But it can help you create a text file viewer--use it to open a file, and read it in to a multi-line edit or list box control!

ASCII:Supported Data Types

STRING
GROUP

ASCII:File Specifications/Maximums

File Size:	4 GB
Records per File:	4,294,967,295 bytes
Record Size:	65,520 bytes
Field Size:	65,520 bytes
Fields per Record:	65,520
Keys/Indexes per File:	n/a
Key Size:	n/a
Memo fields per File:	n/a
Memo Field Size:	n/a
Open Data Files:	Operating system dependent

ASCII:Driver Strings

There are switches or "driver strings" you can set to control the way your application creates, reads, and writes files with a specific driver. Driver strings are simply messages or parameters that are sent to the file driver at run-time to control its behavior. See *Common Driver Features--Driver Strings* for an overview of these runtime Database Driver switches and parameters.

Note:

Some driver strings have no effect after the file is open, so no SEND function syntax is listed for those strings. However, the SEND function syntax to return the value of the switch is listed for all driver strings.

The ASCII Driver supports the following Driver Strings:

CLIP

DRIVER('ASCII', '/CLIP = on | off')

[Clip" =] SEND(file, 'CLIP [= on | off]')

The driver automatically removes trailing spaces from a record before writing it to file. Conversely, the driver automatically expands the clipped records with spaces when read. To disable this feature, set CLIP to OFF. The default is ON. SEND returns the CLIP setting (ON or OFF) in the form of a STRING(3).

CTRLZIEOF

DRIVER('ASCII', '/CTRLZIEOF = on | off')

[EOF" =] SEND(file, 'CTRLZIEOF [= on | off]')

By default (CTRLZIEOF=on) the file driver assumes that any Ctrl-Z characters in the file indicate the end of file. To disable this feature set CTRLZIEOF=off. SEND returns the CTRLZIEOF setting in a STRING(3).

ENDOFRECORD

DRIVER('ASCII', '/ENDOFRECORD = n [,m]')

[EOR" =] SEND(file, 'ENDOFRECORD [= n [,m]]')

Specifies the end of record delimiter.

n represents the number of characters that make up the end-of-record delimiter.

m represents the ASCII code(s) for the end-of-record delimiter, separated by commas. The default is 2,13,10, indicating 2 characters mark the end-of-record, namely, carriage return (13) and line feed (10). SEND returns the end of record delimiter.

Tip

Mainframes and MACs frequently use just a carriage return to delimit records. You can use ENDOFRECORD=1,13 to read these files. UNIX/Linux files frequently terminate with just a line feed and can be read using ENDOFRECORD=1,10

FILEBUFFERS

DRIVER('ASCII', '/FILEBUFFERS = n')

[Buffers" =] SEND(file, 'FILEBUFFERS [= n]')

Sets the size of the buffer used to read and write to the file, where the buffer size is ($n * 512$ bytes). Use the /FILEBUFFERS driver string to increase the buffer size if access is slow. Maximum buffer size is 4,294,967,264. SEND returns the size of the buffer in bytes.

Tip

The default buffer size for files opened denying write access to other users is the larger of 1024 or (2 * record size), and the larger of 512 or record size for all other open modes.

TAB

DRIVER('ASCII', '/TAB = n')

[Spaces" =] SEND(file, 'TAB [= n]')

Sets or queries TAB/SPACE expansion. The ASCII driver expands TABs (ASCII character 9) to spaces when reading. The value indicates the number of spaces with which to replace the tab, subject to the guidelines below. The default value is 8. SEND returns the number of spaces which replace the tab character.

If $n > 0$, spaces replace each tab until the character pointer moves to the next multiple of n . For example, with the default of 8, if the TAB character is the third character in the record, 6 spaces replace the TAB.

If $n = 0$, the driver removes tabs without replacement.

If $n < 0$, the driver removes tabs with the positive value of n spaces. For example, "TAB=-4" causes 4 spaces to replace every tab, regardless of the position of the tab in the record.

If $n = -100$, tabs remain as tabs; the driver *does not* replace them with spaces.

QUICKSCAN

DRIVER('ASCII', '/QUICKSCAN = on | off')

[QScan" =] SEND(file, 'QUICKSCAN [= on | off]')

Specifies buffered access behavior. The ASCII driver reads a buffer at a time (not a record), allowing faster access. In a multi-user environment these buffers are not 100% trustworthy for subsequent access, because another user may change the file between accesses. As a safeguard, the driver rereads the buffers before each record access. To *disable* the reread, set QUICKSCAN to ON. The default is ON for files opened denying write access to other users, and OFF for all other open modes. SEND returns the Quickscan setting (ON or OFF) in the form of a STRING(3).

ASCII:Supported Commands and Attributes

File Attributes	Supported
CREATE	Y
DRIVER(<i>filetype</i> [, <i>driver string</i>])	Y
NAME	Y
ENCRYPT	N
OWNER(<i>password</i>)	N
RECLAIM	N
PRE(<i>prefix</i>)	Y
BINDABLE	Y
THREAD	Y ₄
EXTERNAL(<i>member</i>)	Y
DLL(<i>[flag]</i>)	Y
OEM	Y
File Structures	Supported
INDEX	N
KEY	N
MEMO	N
BLOB	N
RECORD	Y
Index, Key, Memo Attributes	Supported
BINARY	N
DUP	N
NOCASE	N
OPT	N
PRIMARY	N

NAME	N
Ascending Components	N
Descending Components	N
Mixed Components	N
Field Attributes	Supported
DIM	Y
OVER	Y
NAME	Y
File Procedures	Supported
BOF(<i>file</i>)	N
BUFFER(<i>file</i>)	N
BUILD(<i>file</i>)	N
BUILD(<i>key</i>)	N
BUILD(<i>index</i>)	N
BUILD(<i>index, components</i>)	N
BUILD(<i>index, components, filter</i>)	N
BYTES(<i>file</i>)	Y
CLOSE(<i>file</i>)	Y
COPY(<i>file, new file</i>)	Y
CREATE(<i>file</i>)	Y
DUPLICATE(<i>file</i>)	N
DUPLICATE(<i>key</i>)	N
EMPTY(<i>file</i>)	Y
EOF(<i>file</i>)	Y
FLUSH(<i>file</i>)	N
LOCK(<i>file</i>)	Y
NAME(<i>label</i>)	Y

OPEN(<i>file, access mode</i>)	Y
PACK(<i>file</i>)	N
POINTER(<i>file</i>)	Y ₂
POINTER(<i>key</i>)	N
POSITION(<i>file</i>)	Y ₃
POSITION(<i>key</i>)	N
RECORDS(<i>file</i>)	N
RECORDS(<i>key</i>)	N
REMOVE(<i>file</i>)	Y
RENAME(<i>file, new file</i>)	Y
SEND(<i>file, message</i>)	Y
SHARE(<i>file, access mode</i>)	Y
STATUS(<i>file</i>)	Y
STREAM(<i>file</i>)	N
UNLOCK(<i>file</i>)	Y
Record Access	Supported
ADD(<i>file</i>)	Y
ADD(<i>file, length</i>)	N
APPEND(<i>file</i>)	Y
APPEND(<i>file, length</i>)	N
DELETE(<i>file</i>)	N
GET(<i>file, key</i>)	N
GET(<i>file, filepointer</i>)	Y
GET(<i>file, filepointer, length</i>)	N
GET(<i>key, keypointer</i>)	N
HOLD(<i>file</i>)	N
NEXT(<i>file</i>)	Y

NOMEMO(<i>file</i>)	N
PREVIOUS(<i>file</i>)	N
PUT(<i>file</i>)	Y ₁
PUT(<i>file</i> , <i>filepointer</i>)	Y ₁
PUT(<i>file</i> , <i>filepointer</i> , <i>length</i>)	N
RELEASE(<i>file</i>)	N
REGET(<i>file</i> , <i>string</i>)	Y
REGET(<i>key</i> , <i>string</i>)	N
RESET(<i>file</i> , <i>string</i>)	Y
RESET(<i>key</i> , <i>string</i>)	N
SET(<i>file</i>)	Y
SET(<i>file</i> , <i>key</i>)	N
SET(<i>file</i> , <i>filepointer</i>)	Y
SET(<i>key</i>)	N
SET(<i>key</i> , <i>key</i>)	N
SET(<i>key</i> , <i>keypointer</i>)	N
SET(<i>key</i> , <i>key</i> , <i>filepointer</i>)	N
SKIP(<i>file</i> , <i>count</i>)	N
WATCH(<i>file</i>)	N
Transaction Processing	Supported
LOGOUT(<i>timeout</i> , <i>file</i> , ..., <i>file</i>)	N
COMMIT	N
ROLLBACK	N
Null Data Processing	Supported
NULL(<i>field</i>)	N
SETNULL(<i>field</i>)	N
SETNONNULL(<i>field</i>)	N

Notes

- 1 When using PUT() with this driver you should take care to PUT back the same number of characters that were read. If you PUT back more characters than were read, then the "extra" characters will overwrite the first part of the subsequent record. If you PUT back fewer characters than were read, then only the first part of the retrieved record is overwritten, while the last part of the retrieved record remains as it was prior to the PUT().
- 2 POINTER() returns the relative byte position within the file.
- 3 POSITION(file) returns a STRING(4).
- 4 THREADED files consume additional file handles for each thread that accesses the file.

Basic Database Driver

Basic:Specifications

The BASIC file driver reads and writes comma-delimited ASCII files. By default, quotes (" ") surround strings, commas delimit fields, and a carriage-return/line-feed delimits records. The original BASIC programming language defined this file format. The Basic driver does not support keys or backward file processing (thus Basic files are not a good choice for random access processing).

Tip

The Basic file format provides a good choice for a common file format for sharing data with spreadsheet programs. A common file extension used for these files is *.CSV, which stands for "comma separated values."

Files:	C60BASXL.LIB	Windows Static Link Library (32-bit)
	C60BASX.LIB	Windows Export Library (32-bit)
	C60BASX.DLL	Windows Dynamic Link Library (32-bit)

Basic:Supported Data Types

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

Basic:File Specifications/Maximums

File Size:	4 GB
Records per File:	4,294,967,295 bytes
Record Size:	65,520 bytes
Field Size:	65,520 bytes
Fields per Record:	65,520
Keys/Indexes per File:	n/a
Key Size:	n/a
Memo fields per File:	0
Memo Field Size:	n/a
Open Data Files:	Operating system dependent

Basic:Driver Strings

There are switches or "driver strings" you can set to control the way your application creates, reads, and writes files with a specific driver. Driver strings are simply messages or parameters that are sent to the file driver at run-time to control its behavior. See *Common Driver Features--Driver Strings* for an overview of these runtime Database Driver switches and parameters.

Note:

Some driver strings have no effect after the file is open, so no SEND function syntax is listed for those strings. However, the SEND function syntax to return the value of the switch is listed for all driver strings.

The Basic Driver supports the following Driver Strings:

ALWAYSQUOTE

DRIVER('BASIC', '/ALWAYSQUOTE = on | off')

[QScan" =] SEND(file, 'ALWAYSQUOTE [= on | off]')

For compatibility with Basic format data files created by products that do *not* place string values in quotes, set ALWAYSQUOTE to OFF.

When the contents of a string field includes the comma or quote character(s), and ALWAYSQUOTE is off, the Basic driver automatically places quotes around the string when writing to file. This also applies to delimiter characters specified with FIELDDELIMITER, or COMMA. For example, with the defaults in use and ALWAYSQUOTE off, a STRING field containing the value *1313 Mockingbird Lane, Apt. 33* is automatically stored as: "1313 Mockingbird Lane, Apt. 33"

SEND returns the ALWAYSQUOTE setting (ON or OFF) in the form of a STRING(3).

COMMA

DRIVER('BASIC', '/COMMA = n')

[Comma" =] SEND(file, 'COMMA [= n]')

Specifies a single character field separator.

n represents the ANSI code for the field separator character. The default is 44, which is equivalent to "/FIELDDELIMITER=1,44."

SEND returns the ASCII code for the field separator character.

CTRLZISEOF

```
DRIVER('BASIC', '/CTRLZISEOF = on | off' )
```

```
[ EOF" = ] SEND(file, 'CTRLZISEOF [ = on | off ]' )
```

By default (CTRLZISEOF=on) the file driver assumes that any Ctrl-Z characters in the file indicate the end of file. To disable this feature set CTRLZISEOF=off. SEND returns the CTRLZISEOF setting in a STRING(3).

ENDOFRECORD

```
DRIVER('BASIC', '/ENDOFRECORD = n [,m ]' )
```

```
[ EOR" = ] SEND(file, 'ENDOFRECORD [ = n [,m ]]' )
```

Specifies the end of record delimiter.

n represents the number of characters that make up the end-of-record delimiter.

m represents the ANSI code(s) for the end-of-record delimiter, separated by commas. The default is 2,13,10, indicating 2 characters mark the end-of-record, namely, carriage return (13) and line feed (10). SEND returns the end of record delimiter.

Tip

Mainframes frequently use just a carriage return to delimit records. You can use ENDOFRECORD to read these files.

ENDOFRECORDINQUOTE

```
DRIVER('BASIC', '/ENDOFRECORDINQUOTE = on | off' )
```

```
[ EORQuote" = ] SEND(file, 'ENDOFRECORDINQUOTE [ = on | off ]' )
```

By default (ENDOFRECORDINQUOTE=ON) the file driver does not recognize an end-of-record marker that is within a quoted string. To force End-Of-Record markers to always terminate a record, set ENDOFRECORDINQUOTE=OFF. SEND returns the ENDOFRECORDINQUOTE setting (ON or OFF) in the form of a STRING(3).

FILEDDELIMITER

`DRIVER('BASIC', '/FILEDDELIMITER = n [,m]')`

`[Limiter" =] SEND(file, 'FILEDDELIMITER [= n [,m]]')`

Specifies the field separator. This is in addition to any string delimiter specified by the /QUOTE driver string.

n represents the number of characters that make up the field separator.

m represents the ANSI code(s) for the field separator characters, separated by commas. The default is 1,44 which indicates the comma character.

SEND returns the field delimiter character.

Tip

If both FILEDDELIMITER and COMMA are specified, the last specification prevails.

FILEBUFFERS

`DRIVER('BASIC', '/FILEBUFFERS = n')`

`[Buffers" =] SEND(file, 'FILEBUFFERS [= n]')`

Sets the size of the buffer used to read and write to the file, where the buffer size is (*n* * 512 bytes). Use the /FILEBUFFERS driver string to increase the buffer size if access is slow. Maximum buffer size is 4,294,967,264. SEND returns the size of the buffer in bytes.

Tip

The default buffer size for files opened denying write access to other users is the larger of 1024 or (2 * record size), and the larger of 512 or record size for all other open modes.

QUICKSCAN

`DRIVER('BASIC', '/QUICKSCAN = on | off')`

`[QScan" =] SEND(file, 'QUICKSCAN [= on | off]')`

Specifies buffered access behavior. The ASCII driver reads a buffer at a time (not a record), allowing faster access. In a multi-user environment these buffers are not 100% trustworthy for subsequent access, because another user may change the file between accesses. As a safeguard, the driver rereads the buffers before each record access. To *disable* the reread, set QUICKSCAN to ON. The default is ON for files opened denying write access to other users, and OFF for all other open modes. SEND returns the QUICKSCAN setting (ON or OFF) in the form of a STRING(3).

Tip

TAB-delimited values are a common format compatible with the Windows clipboard. Using the BASIC file driver string /COMMA=9 lets you read Windows clipboard files

QUOTE

DRIVER('BASIC', '/QUOTE = n')

[Quote" =] SEND(file, 'QUOTE [= n]')

Specifies a single character string delimiter.

n is the ANSI code of the delimiter character. The default is 34, the ASCII value for the quotation mark.

SEND returns the ASCII code of the single character string delimiter.

Popular File Formats

The following demonstrates how to use the driver strings to create two popular file formats:

- † Microsoft Word for Windows Mail Merge:

```
/ALWAYSQUOTE=OFF
/FIELDDDELIMITER=1,9
/ENDOFRECORD=1,13
```

- † TAB delimited format:

```
/COMMA=9
```

Basic:Supported Commands and Attributes

File Attributes	Supported
CREATE	Y
DRIVER(<i>filetype</i> [, <i>driver string</i>])	Y
NAME	Y
ENCRYPT	N
OWNER(<i>password</i>)	N
RECLAIM	N
PRE(<i>prefix</i>)	Y
BINDABLE	Y
THREAD	Y ⁴
EXTERNAL(<i>member</i>)	Y
DLL(<i>flag</i>)	Y
OEM	Y
File Structures	Supported
INDEX	N
KEY	N
MEMO	N
BLOB	N
RECORD	Y
Index, Key, Memo, Attributes Supported	
BINARY	N
DUP	N
NOCASE	N
OPT	N
PRIMARY	N

NAME	N
Ascending Components	N
Descending Components	N
Mixed Components	N
Field Attributes	Supported
DIM	Y
OVER	Y
NAME	Y
File Procedures	Supported
BOF(<i>file</i>)	N
BUFFER(<i>file</i>)	N
BUILD(<i>file</i>)	N
BUILD(<i>key</i>)	N
BUILD(<i>index</i>)	N
BUILD(<i>index, components</i>)	N
BUILD(<i>index, components, filter</i>)	N
BYTES(<i>file</i>)	Y
CLOSE(<i>file</i>)	Y
COPY(<i>file, new file</i>)	Y
CREATE(<i>file</i>)	Y
DUPLICATE(<i>file</i>)	N
DUPLICATE(<i>key</i>)	N
EMPTY(<i>file</i>)	Y
EOF(<i>file</i>)	Y
FLUSH(<i>file</i>)	N
LOCK(<i>file</i>)	Y
NAME(<i>label</i>)	Y

OPEN(<i>file, access mode</i>)	Y
PACK(<i>file</i>)	N
POINTER(<i>file</i>)	Y ₂
POINTER(<i>key</i>)	N
POSITION(<i>file</i>)	Y ₃
POSITION(<i>key</i>)	N
RECORDS(<i>file</i>)	N
RECORDS(<i>key</i>)	N
REMOVE(<i>file</i>)	Y
RENAME(<i>file, new file</i>)	Y
SEND(<i>file, message</i>)	Y
SHARE(<i>file, access mode</i>)	Y
STATUS(<i>file</i>)	Y
STREAM(<i>file</i>)	N
UNLOCK(<i>file</i>)	Y
Record Access	Supported
ADD(<i>file</i>)	Y
ADD(<i>file, length</i>)	N
APPEND(<i>file</i>)	Y
APPEND(<i>file, length</i>)	N
DELETE(<i>file</i>)	N
GET(<i>file, key</i>)	N
GET(<i>file, filepointer</i>)	Y
GET(<i>file, filepointer, length</i>)	N
GET(<i>key, keypointer</i>)	N
HOLD(<i>file</i>)	N
NEXT(<i>file</i>)	Y

NOMEMO(<i>file</i>)	N
PREVIOUS(<i>file</i>)	N
PUT(<i>file</i>)	Y ₁
PUT(<i>file</i> , <i>filepointer</i>)	Y ₁
PUT(<i>file</i> , <i>filepointer</i> , <i>length</i>)	N
RELEASE(<i>file</i>)	N
REGET(<i>file</i> , <i>string</i>)	Y
REGET(<i>key</i> , <i>string</i>)	N
RESET(<i>file</i> , <i>string</i>)	Y
RESET(<i>key</i> , <i>string</i>)	N
SET(<i>file</i>)	Y
SET(<i>file</i> , <i>key</i>)	N
SET(<i>file</i> , <i>filepointer</i>)	Y
SET(<i>key</i>)	N
SET(<i>key</i> , <i>key</i>)	N
SET(<i>key</i> , <i>keypointer</i>)	N
SET(<i>key</i> , <i>key</i> , <i>filepointer</i>)	N
SKIP(<i>file</i> , <i>count</i>)	N
WATCH(<i>file</i>)	N
Transaction Processing	Supported
LOGOUT(<i>timeout</i> , <i>file</i> , ..., <i>file</i>)	N
COMMIT	N
ROLLBACK	N
Null Data Processing	Supported
NULL(<i>field</i>)	N
SETNULL(<i>field</i>)	N
SETNONNULL(<i>field</i>)	N

Notes

- 1 When using PUT() with this driver you should take care to PUT back the same number of characters that were read. If you PUT back more characters than were read, then the "extra" characters will overwrite the first part of the subsequent record. If you PUT back fewer characters than were read, then only the first part of the retrieved record is overwritten, while the last part of the retrieved record remains as it was prior to the PUT().
- 2 POINTER() returns the relative byte position within the file.
- 3 POSITION(file) returns a STRING(4).
- 4 THREADED files consume additional file handles for each thread that accesses the file.

Btrieve Database Driver

Btrieve:Specifications

This file driver reads and writes Btrieve files using low-level direct access.

Under Clarion, the Btrieve file driver is implemented by using .DLLs and an .EXE supplied by Pervasive Software (formerly Btrieve Technologies, Inc.). For an application to use a Btrieve file driver, the following files must accompany the executable:

You must purchase a 32-bit Btrieve engine from Pervasive Software.

LICENSE WARNING: A registered Clarion owner cannot redistribute the above files outside of his/her organization without a license from Pervasive Software.

Files:	C60BTRXL.LIB	Windows Static Link Library
	C60BTRX.LIB	Windows Export Library
	C60BTRX.DLL	Windows Dynamic Link Library

Btrieve:Data Types

<u>Clarion data type</u>	<u>Btrieve data type</u>
BYTE	STRING (1 byte)
SHORT	INTEGER (2 bytes)
LONG	INTEGER (4 bytes)
SREAL	FLOAT (4 bytes)
REAL	FLOAT (8 bytes)
BFLOAT4	BFLOAT (4 bytes)
BFLOAT8	BFLOAT (8 bytes)
PDECIMAL	DECIMAL
STRING	STRING
CSTRING	ZSTRING
PSTRING	LSTRING
DATE	DATE
TIME	TIME
USHORT	UNSIGNED BINARY (2 bytes)
ULONG	UNSIGNED BINARY (4 bytes)
MEMO	STRING,LVAR or NOTE (see below)
BYTE,NAME('LOGICAL')	LOGICAL*
USHORT,NAME('LOGICAL')	LOGICAL*
PDECIMAL,NAME('MONEY')	MONEY*
STRING(@N0n-),NAME('STS')	SIGNED TRAILING SEPARATE*
DECIMAL*	

Notes:

- * You can store Clarion DECIMAL types in a Btrieve file. However, you cannot build a key or index using the field. This is provided for backward compatibility with older Clarion programs which used the Btrieve LEM. If you need standard Btrieve decimal data that is compatible with any other Btrieve compliant program, you should use the PDECIMAL data type and not the DECIMAL data type.
- * If you want to create a file with LOGICAL or MONEY field types, you must specify LOGICAL or MONEY in the field's NAME attribute. If you are accessing an existing file, the NAME attribute is not required.

LOGICAL may be declared as a BYTE or USHORT, depending on whether it is a one or two byte LOGICAL:

```
LogicalField1  BYTE      !One byte LOGICAL
LogicalField2  USHORT    !Two byte LOGICAL
```

MONEY may be declared as a PDECIMAL(x,2), where x is the total number of digits to be stored:

```
MoneyField  PDECIMAL(7,2),NAME('MONEY') !Store up to 99999.99
```

- * Btrieve NUMERIC fields are not fully supported by the driver. Btrieve NUMERIC is stored as a string with the last character holding a digit and an implied sign. The possible values for this last character are:

```

          1 2 3 4 5 6 7 8 9 0
Positive:  A B C D E F G H I {
Negative:  J K L M N O P Q R }
```

To access a NUMERIC field you must define a STRING(@N0x), where x is one less than the digits in the NUMERIC, and a STRING(1) to hold the sign indicator. The Btrieve driver does not maintain this sign field, the application must be written to directly handle it.

For example to access a NUMERIC(7) you would have:

```

NumericGroup  GROUP          !Store -999999 to 999999
Number        STRING(@N06) !Numbers
Sign          STRING(1)     !Sign indicator
END
```

Btrieve:File Specifications/Maximums

```

File Size                : 4,000,000,000 bytes
Records per File        : Limited by the size of the file
Record Size:
  Client-based          : 65,535 bytes variable length
  Server based         : 57,000 bytes variable length
Field Size              : 65,520 bytes
Fields per Record      : 65,520
Keys/Indexes per File  : 24 with NLM5
                       : 256 with NLM6.
                       : Client Btrieve v6.15
                       : Page Size Max Key Segments
                       : 512      8
                       : 1,024   23
                       : 1,536   24
                       : 2,048   54
                       : 4,096   119

This is the total number of components. If you have a
multicomponent key built from three fields, this counts
as three indexes when counting the number of allowed
indexes.
Key Size                 : 255 bytes

Memo fields per File    : System memory dependent
Memo field size        : 65,520 bytes
Open Files              : Operating system dependent

```

Note:

The Btrieve driver supports data only and key only files.

Btrieve:Driver Strings

There are switches or "driver strings" you can set to control the way your application creates, reads, and writes files with a specific driver. Driver strings are simply messages or parameters that are sent to the file driver at run-time to control its behavior. See Common Driver Features--Driver Strings for an overview of these runtime Database Driver switches and parameters.

The Btrieve Driver supports the following Driver Strings:

ACS

DRIVER('BTRIEVE', '/ACS = filename')

[SortSeq" =] SEND(file, 'ACS [= filename]')

When creating a Btrieve file you can specify an alternate collating sequence for sorting STRING keys. This sorting sequence is normally obtained from the sort sequence you define in the INI file for your program. However, Btrieve supplies files for doing case insensitive sorts. To create your file using these sort sequences you specify the name of the sort file in the driver string.

ALLOWREAD

DRIVER('BTRIEVE', '/ALLOWREAD = ON | OFF')

[Read" =] SEND(file, 'ALLOWREAD [= ON | OFF]')

By default, a Btrieve file created with an owner name may be accessed *only* in read-only mode when the owner name is not known. To prevent *all* access to the file without the owner name, set ALLOWREAD to OFF. SEND returns the ALLOWREAD setting (ON or OFF) in the form of a STRING(3).

APPENDBUFFER

DRIVER('BTRIEVE', '/APPENDBUFFER = size ')

[Buffer" =] SEND(file, 'APPENDBUFFER [= size]')

By default, APPEND adds records to the file one at a time. To get better performance over a network you can tell the driver to build up a buffer of records then send all of them to Btrieve at once. *Size* is the number of records you want to allocate for the buffer. SEND returns the number of records that will fit in the buffer.

BALANCEKEYS

DRIVER('BTRIEVE', '/BALANCEKEYS = ON | OFF')

[Balance" =] SEND(file, 'BALANCEKEYS [= ON | OFF]')

When creating a Btrieve file, you can use this driver string to tell Btrieve that all keys associated with the file must be stored in a balanced btree. This saves disk space, but will slow down file adds, deletes and updates where key values change. SEND returns the BALANCEKEYS setting (ON or OFF) in the form of a STRING(3).

COMPRESS

DRIVER('BTRIEVE', '/COMPRESS = ON | OFF')

[Read" =] SEND(file, 'COMPRESS [= ON | OFF]')

Btrieve lets you compress the data before storage. This allows for a smaller storage requirement, but reduces performance. When COMPRESS is ON, CREATE creates a compressed Btrieve file. SEND returns the COMPRESS setting (ON or OFF) in the form of a STRING(3).

FREESPACE

DRIVER('BTRIEVE', '/FREESPACE = 0 | 10 | 20 | 30')

[Read" =] SEND(file, 'FREESPACE [= 0 | 10 | 20 | 30]')

Specifies the percentage of free space to maintain on variable length pages. The default is zero. SEND returns the percentage of free space to maintain on variable length pages.

LACS

DRIVER('BTRIEVE', '/LACS [= | country_id,codepage]')

[Sequence" =] SEND(file, 'LACS [= | country_id,codepage]')

Btrieve v6.15 and later offers the Local Alternate Collating Sequences feature. This allows your string keys to sort based on the country code for the machine running your program. To use this feature include '/LACS' in your driver string.

/LACS=country_ID,code_page

You can also specify a User-Defined Alternate Collating Sequence. This allows your string key to sort based on the DOS country code and code page for a particular country. To use this feature you put '/LACS=country_id,codepage' in your driver string. Note that there must be no spaces surrounding the comma.

SEND returns country_id,codepage or the string ',' if using the Local Alternate Collating Sequences feature.

MEMO

DRIVER('BTRIEVE', '/MEMO = SINGLE | LVAR | NOTE [,delimiter]')

[Memo" =] SEND(file, 'MEMO [= SINGLE | LVAR | NOTE [,delimiter]]')

/MEMO=SINGLE

To access existing Btrieve files created with the Btrieve LEM from Clarion Professional Developer 2.1(DOS), or files with variable length records set MEMO to SINGLE.

To access a file with variable length records, use a SINGLE style MEMO whose size equals the maximum of the variable length component of the record. To add/put records to this style file with binary data stored in variable length section, use the ADD(file,length), APPEND(file,length) and PUT(file,pos,length) functions. The driver ignores the *pos* parameter in the PUT function, but initialize it to 0 (zero) for future compatibility. The APPEND or PUT functions will remove all trailing spaces for text memos and NULL characters for binary records before storing the record.

/MEMO=LVAR**/MEMO=NOTE,<delimiter>**

To access Xtrieve data files that have data type of Note or LVar, set the driver string to NOTE and LVAR respectively. With the NOTE data type, specify the end-of-field delimiter. Specify the ASCII value for the delimiter. NOTE and LVAR memos do not require the use of the size variants of ADD, APPEND and PUT, when storing records. The end of record marker is not necessary for a NOTE style memo. The driver automatically adds the end of record marker before storing the record and removes it before putting the memo data into the memo buffer.

As an example, "/MEMO=NOTE,141" indicates a file with an Xtrieve Notes field using CR/LF as the delimiter. For more information on the Xtrieve data types refer to the documentation supplied by Novell.

SEND(file,'MEMO')

Returns the MEMO setting: NORMAL, NOTE, LVAR, or SINGLE.

PAGESIZE

DRIVER('BTRIEVE', '/PAGESIZE = SIZE')

[PSize" =] SEND(file, 'PAGESIZE[=SIZE')

Sets the Btrieve Page size at file creation time. The size must be a multiple of 512, with a maximum of 4096. Larger page sizes usually result in more efficient disk storage.

SEND returns the page size setting.

PREALLOCATE

DRIVER('BTRIEVE', '/PREALLOCATE = n')

[Read" =] SEND(file, 'PREALLOCATE [= n]')

When creating a Btrieve file, you can preallocate *n* pages of disk space for the file. The default is zero. SEND returns the number of pages of allocated disk space.

TRUNCATE

DRIVER('BTRIEVE', '/TRUNCATE = ON | OFF')

[Trunc" =] SEND(file, 'TRUNCATE [= ON | OFF]')

When creating a Btrieve file, you can use this driver string to tell Btrieve to truncate trailing spaces. This forces the record to be stored as a variable length record. SEND returns the TRUNCATE setting (ON or OFF) in the form of a STRING(3).

Btrieve:Driver Properties

You can use Clarion's property syntax to query and set certain driver properties. These properties are described below.

PROP:PageLevelLocking

PROP:PageLevelLocking sets the type of locking the driver uses with LOGOUT. The driver uses either page or file level locking schemes. Set to PageLevelLocking by setting the property to '1'. This is the default. To set the driver to file level locking, set the property to ''.

```
MyFile{PROP:PageLevelLocking} = '1'      !Set Page level locking
MyFile{PROP:PageLevelLocking} = ''      !Set File level locking
loc:lock = MyFile{PROP:PageLevelLocking} !read locking scheme
```

PROP:PositionBlock

PROP:PositionBlock returns the Btrieve pointer to the Btrieve position block used by the Btrieve driver for the named file. This allows you to call Btrieve operations directly. For example:

```
MAP
MODULE('Btrieve')
  BTRV(USHORT, LONG, <*STRING>, *UNSIGNED, <*STRING>, BYTE, BYTE), |
  NAME('BTRV'), PASCAL, RAW
END
END

StatData STRING(33455)
KeyData  STRING(64)
DataLen  UNSIGNED(33455)

CODE
PosBlock = file{PROP:PositionBlock}
BTRV(15, PosBlock, StatData, DataLen, KeyData, 64, 0) !Get file statistics
```

Btrieve:Supported Commands and Attributes

File Attributes Supported

CREATE	Y
DRIVER(<i>filetype</i> [, <i>driver string</i>])	Y
NAME	Y
ENCRYPT	Y
OWNER(<i>password</i>)	Y ₁
RECLAIM	Y
PRE(<i>prefix</i>)	Y
BINDABLE	Y
THREAD	Y ₁₅
EXTERNAL(<i>member</i>)	Y
DLL(<i>[flag]</i>)	Y
OEM	Y

File Structures Supported

INDEX	Y
KEY	Y
MEMO	Y ₂
BLOB	N
RECORD	Y

Index, Key, Memo Attributes Supported

BINARY	Y ₁₆
DUP	Y
NOCASE	Y
OPT	Y
PRIMARY	Y

NAME	Y ₂
Ascending Components	Y
Descending Components	Y
Mixed Components	Y
Field Attributes	Supported
DIM	Y
OVER	Y
NAME	Y
File Procedures	Supported
BOF(<i>file</i>)	Y ₁₀
BUFFER(<i>file</i>)	N
BUILD(<i>file</i>)	Y ₃
BUILD(<i>key</i>)	Y ₃
BUILD(<i>index</i>)	Y ₃
BUILD(<i>index, components</i>)	Y ₃
BUILD(<i>index, components, filter</i>)	N
BYTES(<i>file</i>)	N
CLOSE(<i>file</i>)	Y
COPY(<i>file, new file</i>)	Y
CREATE(<i>file</i>)	Y
DUPLICATE(<i>file</i>)	Y
DUPLICATE(<i>key</i>)	Y
EMPTY(<i>file</i>)	Y
EOF(<i>file</i>)	Y ₁₀
FLUSH(<i>file</i>)	Y
LOCK(<i>file</i>)	N ₄
NAME(<i>label</i>)	Y

OPEN(<i>file, access mode</i>)	Y
PACK(<i>file</i>)	Y
POINTER(<i>file</i>)	Y ¹¹
POINTER(<i>key</i>)	Y ¹¹
POSITION(<i>file</i>)	Y ¹²
POSITION(<i>key</i>)	Y ¹²
RECORDS(<i>file</i>)	Y
RECORDS(<i>key</i>)	Y
REMOVE(<i>file</i>)	Y
RENAME(<i>file, new file</i>)	Y
SEND(<i>file, message</i>)	Y
SHARE(<i>file, access mode</i>)	Y
STATUS(<i>file</i>)	Y
STREAM(<i>file</i>)	Y
UNLOCK(<i>file</i>)	N
Record Access	Supported
ADD(<i>file</i>)	Y ⁵
ADD(<i>file, length</i>)	Y ⁵
APPEND(<i>file</i>)	Y ⁶
APPEND(<i>file, length</i>)	Y ^{5,6}
DELETE(<i>file</i>)	Y ⁷
GET(<i>file, key</i>)	Y
GET(<i>file, filepointer</i>)	Y
GET(<i>file, filepointer, length</i>)	N
GET(<i>key, keypointer</i>)	Y
HOLD(<i>file</i>)	Y
NEXT(<i>file</i>)	Y

NOMEMO(<i>file</i>)	Y
PREVIOUS(<i>file</i>)	Y
PUT(<i>file</i>)	Y ⁵
PUT(<i>file</i> , <i>filepointer</i>)	N
PUT(<i>file</i> , <i>filepointer</i> , <i>length</i>)	Y
RELEASE(<i>file</i>)	Y
REGET(<i>file</i> , <i>string</i>)	Y
REGET(<i>key</i> , <i>string</i>)	Y
RESET(<i>file</i> , <i>string</i>)	Y
RESET(<i>key</i> , <i>string</i>)	Y
SET(<i>file</i>)	Y
SET(<i>file</i> , <i>key</i>)	Y
SET(<i>file</i> , <i>filepointer</i>)	Y ⁸
SET(<i>key</i>)	Y
SET(<i>key</i> , <i>key</i>)	Y
SET(<i>key</i> , <i>keypointer</i>)	Y ⁸
SET(<i>key</i> , <i>key</i> , <i>filepointer</i>)	Y ⁹
SKIP(<i>file</i> , <i>count</i>)	Y
WATCH(<i>file</i>)	Y
Transaction Processing	Supported
LOGOUT(<i>timeout</i> , <i>file</i> , ..., <i>file</i>)	Y ^{13,14}
COMMIT	Y ¹⁴
ROLLBACK	Y
Null Data Processing	Supported
NULL(<i>field</i>)	N
SETNULL(<i>field</i>)	N
SETNONNULL(<i>field</i>)	N

Notes

- 1 We recommend using a variable password that is lengthy and contains special characters because this more effectively hides the password value from anyone looking for it. For example, a password like "dd...#\$...*&" is much more difficult to "find" than a password like "SALARY."

Tip

To specify a variable instead of the actual password in the Owner Name field of the File Properties dialog, type an exclamation point (!) followed by the variable name. For example: !MyPassword.

- 2 The driver ignores any NAME attribute on a MEMO field. MEMO fields can reside either in a separate file, or in the data file if the driver string /MEMO is set to SINGLE, LVAR or NOTE. If the driver string /MEMO is not set, the separate MEMO file name is "MEM," preceded by the first five characters of the file's label, plus the file extension ".DAT." Setting the driver string /MEMO restricts you to one memo field per file.
- 3 If used after an APPEND(), but before a file is closed, this adds the keys dropped by APPEND(). In all other cases BUILD() rebuilds the file and keys. If you only want to rebuild keys, doing a BUILD(key) for each key is faster than BUILD(file).
- 4 Btrieve does not directly support file locking. If you require file locking, use LOGOUT.
- 5 When using the LVAR and NOTE memo type, make certain that the memo has the appropriate structure. If the structure is incorrect and the driver calculates a length greater than the maximum memo size defined for that file, these functions fail and set errorcode to 57 - Invalid Memo File.
- 6 Btrieve does not support non-key updates. To emulate APPEND() behavior, the driver drops all indexes possible when APPEND() is first called. Calling BUILD() immediately after appending records rebuilds the dropped key fields.
- 7 Btrieve's DELETE destroys positioning information when processing in file order. The driver attempts to reposition to the appropriate record. This is not always possible and may require the driver to read from the start of the file. Using key order processing avoids this possible slow down.
- 8 If a file pointer or key pointer has a value of zero, the driver ignores the pointer parameter. Processing is set to either file or key order, and the record pointer is set to the first element.
- 9 If the file pointer has a value of zero, processing starts at the first key value whose position is greater than (or less than for PREVIOUS) the file pointer. Not passing a valid pointer, other than maximum LONG or maximum ULONG, is inefficient.

- 10 These functions are supported, but not recommended. They cause more disk I/O than `ERRORCODE()`. `Btrieve` returns `eof` when reading past the last record. Therefore, the driver must read the next record, then the `next` to see if it's at the end of file, then return to the record you want.
- 11 `POINTER()` returns a relative position within the file, not a record number.
- 12 `POSITION(file)` returns a `STRING(4)`. `POSITION(key)` returns a `STRING` the size of the key fields + 4 bytes.
- 13 If a system crashes during a transaction (`LOGOUT--COMMIT`), the recovery is automatically handled by the `Btrieve` driver the next time the affected file is accessed.

When you issue a `LOGOUT`, all `Btrieve` files accessed during the transaction are logged out. The following code is illegal because you cannot close a logged-out file:

```
LOGOUT(1, file1)
OPEN(file1)
CLOSE(file1)
```

- 14 See also `PROP:Logout` in the *Language Reference*.
- 15 `THREADED` files do not consume additional file handles for each thread that accesses the file.
- 16 OEM conversion is not applied to `BINARY MEMOs`. The driver assumes `BINARY MEMOs` are zero padded; otherwise, space padded.

Btrieve:Other

Client/Server

For Client/Server-based `Btrieve`, Netware `Btrieve` is a server-based version of `Btrieve` that runs on a Novell server.

File Structure

A single file normally holds the data and all keys. Data filenames default to a `*.DAT` file extension. By default, the driver stores memos in a separate file, or optionally in the data file itself, given the appropriate driver string.

Because `Btrieve` is a data-model independent, indexed record manager, it does not store field definitions within the data itself. The application accessing the data determines how to interpret the `Btrieve` record. Absent `.DDF` files describing the `Btrieve` file, it is very difficult for an application that does not create or maintain the file to meaningfully interpret its data.

The `Btrieve` file format stores minimal file structure information in the file. As far as possible, the driver validates your description against the information in the file. However, it *is* possible to successfully open a `Btrieve` file that has key definitions that do not exactly match your definition. You must make certain that your file and key definitions accurately match the `Btrieve` file.

Keys and Indexes

KEYs are dynamic, and automatically update when the data file changes.

INDEXes are stored separately from data files. INDEX files receive a temporary file name, and are deleted when the program terminates normally. INDEXes are static--they are not automatically updated when the data file changes. The BUILD statement creates or updates index files.

Record Lengths

The driver stores records less than 4K as fixed length. It stores records greater than 4K as variable length. The minimum record length is 4 bytes. One record can be held in each open file by each user.

Page Size

To determine the physical record length, add 8 bytes for each KEY that allows duplicates. Add 4 bytes if the file allows variable record lengths. Finally, allow 6 bytes for overhead per page.

For example: If the record size is 300 bytes and the file has three KEYs that allow Duplicates, the total record size is:

$$\begin{array}{rcl} & 300 & \text{record size} \\ \times & 24 & \text{overhead for three KEYs with the DUP attribute} \\ = & 324 & \text{physical record length} \end{array}$$

A page size of 512 would only hold one such record, and 182 bytes per page would go unused ($512 - 6 - 324$). If the page size were 1024, three records could be stored per page and only 46 bytes would go unused ($1024 - 6 - (324 * 3)$).

You must load BTRIEVE.EXE with a page size equal to or greater than the largest page size of any file that you will be accessing.

File Sharing

Btrieve lets you open a file in five different formats: NORMAL, ACCELERATED, READ-ONLY, VERIFY, or EXCLUSIVE. The equivalent Clarion OPEN() states are:

<u>Btrieve State</u>	Clarion OPEN/SHARE access mode
ACCELERATED	Read/Write with FCB compatibility mode (2H)
READ-ONLY	Read Only (0H,10H,20H,30H,40H)
VERIFY	Write Only with FCB compatibility mode (1H)
EXCLUSIVE	Write Only with any Deny flag (11H,21H,31H,41H)
	Read/Write with Deny All, Read or Write (12H,22H)
NORMAL	Read/Write with Deny None (42H)

Btrieve allows a file to have a specified owner. See the /READONLY driver string for details on setting this flag. The file may also be encrypted with the ENCRYPT attribute. A file can only be encrypted when an owner name is supplied.

Record Pointers

Btrieve uses an unsigned long for its internal record pointer; negative values are stripped of their sign. We recommend the ULONG data type for your record pointer.

Collating Sequence

- † Key Attribute: **NOCASE**

NLM 5 does not support case insensitive indexing. When necessary, you must supply an alternate collating sequence which implements case insensitive sorting.

Btrieve supports an alternate collating sequence. However, NLM 6 does not support *both* NOCASE *and* an alternate collating sequence. If you specify both, the NOCASE attribute takes precedence. No error is returned from The SEND function.

- † Changing the Collating Sequence

Btrieve stores the collating sequence inside the file. So to change the collating sequence you have to change the .ENV file, then create a new Btrieve file based on the modified .ENV file, then copy the data from the old file to the new file.

KEY Definitions

- † When defining a file, the key definition does not need to exactly match the underlying file. For example, you can have a physical file with a single component STRING(20). You can define this as a key with two string components with a total length of 20. The rule is that the data types must match and the total size must match. However, if your Clarion definition does not exactly match the underlying file, the driver cannot optimize APPEND() or BUILD() statements.
- † A Key's NAME attribute can add additional functionality.

KEY,NAME('MODIFIABLE=true|false')

Btrieve lets you create a key that cannot be changed once created. To use this feature you can use the name attribute on the key to set MODIFIABLE to FALSE. It defaults to TRUE.

KEY,NAME('ANYNULL')

Btrieve lets you create a key that will not include a record if any key components are null. To create such a key you specify ANYNULL in the key name.

For example, to create a key that is non-modifiable and excludes keys if any component is null:

```
Key1 KEY(+pre:field1,-pre:field2),NAME('ANYNULL MODIFIABLE=FALSE')
```

KEY,NAME('AUTOINCREMENT')

The Clarion CREATE statement creates a Btrieve autoincrement key.

KEY,NAME('REPEATINGDUPLICATE')

By default Btrieve version 6 stores a reference to only the first record in a series of duplicate records in a key. The other occurrences of the duplicate key value are obtained by following a link list stored at the record. To create an index where all duplicate records are stored in the key you use the NAME('REPEATINGDUPLICATE'). This produces larger keys, but random access to duplicate records is faster (this feature is only available for version 6 files).

Clarion Database Driver

Clarion:Specifications

The Clarion file driver is compatible with the file system used by Clarion for DOS 3.1 and Clarion Professional Developer 2.1, patch 2.109 and later.

Keys and Indexes exist as separate files from the data file. Keys are dynamic--they are automatically updated as the data file changes. The default file extension for a key file is *.K##. Indexes are static--they do not automatically update, but instead require the BUILD statement for updating.

The driver stores records as fixed length. It stores memo fields in a separate file. The memo file defaults to the first eight characters of the File Label plus an extension of .MEM.

Files:	C60CLAXL.LIB	Windows Static Link Library
	C60CLAX.LIB	Windows Export Library
	C60CLAX.DLL	Windows Dynamic Link Library

Tip

By avoiding the ASCII-only file formats of many other popular PC database application development systems, the Clarion file format provides a more secure means of storing data.

Clarion:Data Types

```

BYTE      DECIMAL1
SHORT    STRING (255 byte maximum)
LONG     MEMO
REAL     GROUP

```

- 1 Decimal sizes greater than 15 are not supported by Clarion Professional Developer 2.1.

Clarion:File Specifications/Maximums

```

File Size:           limited only by disk space
Records per File :  4,294,967,295
Record Size:        65,520 bytes
Field Size :        65,520 bytes
Field Name:         12 characters
Fields per Record:  65,520
Keys/Indexes per File: 251
Key Size:           245 bytes
Memo fields per File : 1
Memo Field Size:    65,520 bytes
Open Data Files:    Operating system dependent

```

Clarion:Driver Strings

There are switches or "driver strings" you can set to control the way your application creates, reads, and writes files with a specific driver. Driver strings are simply messages or parameters that are sent to the file driver at run-time to control its behavior. See *Common Driver Features--Driver Strings* for an overview of these runtime Database Driver switches.

Note:

Some driver strings have no effect after the file is open, so no SEND function syntax is listed for those strings. However, the SEND function syntax to return the value of the switch is listed for all driver strings.

The Clarion Driver supports the following Driver Strings:

DELETED

SEND(file, 'DELETED')

For use only with the SEND command when IGNORESTATUS is on. Reports the status of the loaded record. If deleted, the return string is "ON" and if not "OFF."

HELD

SEND(file, 'HELD')

For use only with the SEND command when IGNORESTATUS is on. Reports the status of the loaded record. If held, the return string is "ON" and if not "OFF."

IGNORESTATUS

DRIVER('Clarion', '/IGNORESTATUS = on | off')

[Status" =] SEND(file, 'IGNORESTATUS [= on | off]')

When set *on*, the driver does *not* skip deleted records when accessing the file with GET(), NEXT(), and PREVIOUS() in file order. It also enables a PUT() on a deleted or held record. IGNORESTATUS requires opening the file in exclusive mode. However, any MEMO data of the deleted records is not recoverable. SEND returns the IGNORESTATUS setting (ON or OFF) in the form of a STRING(3).

MAINTAINHEADERTIME

DRIVER('Clarion', '/MAINTAINHEADERTIME = on | off')

[Status" =] SEND(file, 'MAINTAINHEADERTIME [= on | off]')

When set *on*, the driver maintains the file header time stamp (last updated) under all circumstances. When set to *off* (the default), the driver improves performance by ignoring the time stamp under some circumstances. SEND returns the MAINTAINHEADERTIME setting (ON or OFF) in the form of a STRING(3).

RECOVER

SEND(file, 'RECOVER = n')

The RECOVER string, when *n* is greater than 0, UNLOCKS data files, RELEASEs held records, and rolls back incomplete transactions in order to recover from a system crash. See also *Transaction Processing for Clarion Files*.

n represents the number of seconds to wait before invoking the recovery process. When *n* is equal to 1, the recovery process is invoked immediately. When *n* is equal to 0, the recovery process is disarmed.

There are two ways of using RECOVER:

```
SEND(file,RECOVER=n)
OPEN(file)
```

This releases a LOCK on a file that was locked when a machine crashed. It also rolls back a transaction that was in process when a system crashed.

```
SEND(file,RECOVER=n)
GET or NEXT or PREVIOUS
```

This removes a hold flag from records that were held when a machine crashed. Here is a piece of code that removes all hold flags from a file:

```
OPEN(file)           !make sure no one else is using the file
SEND(file,'IGNORESTATUS=ON')
SET(file)
LOOP
  NEXT(file)
  IF ERRORCODE()
    BREAK
  END
  IF SEND(file,'HELD') = 'ON' THEN
    SEND(file,'RECOVER=1')
    REGET(file,POSITION(file))
  END
END
```

RECOVER may not be used as a DRIVER string--you may only use it with the SEND function. The SEND function returns a blank string.

Clarion:Supported Commands and Attributes

File Attributes	Supported
CREATE	Y
DRIVER(<i>filetype</i> [, <i>driver string</i>])	Y
NAME	Y
ENCRYPT	Y
OWNER(<i>password</i>)	Y ₁
RECLAIM	Y
PRE(<i>prefix</i>)	Y
BINDABLE	Y
THREAD	Y ₈
EXTERNAL(<i>member</i>)	Y
DLL(<i>flag</i>)	Y
OEM	Y

File Structures	Supported
INDEX	Y
KEY	Y
MEMO	Y
BLOB	N
RECORD	Y

Index, Key, Memo Attributes Supported

BINARY	Y ₉
DUP	Y
NOCASE	Y
OPT	Y

PRIMARY	Y
NAME	Y
Ascending Components	Y
Descending Components	N
Mixed Components	N
Field Attributes	Supported
DIM	Y
OVER	Y
NAME	Y
File Procedures	Supported
BOF(<i>file</i>)	Y ₂
BUFFER(<i>file</i>)	N
BUILD(<i>file</i>)	Y
BUILD(<i>key</i>)	Y
BUILD(<i>index</i>)	Y
BUILD(<i>index, components</i>)	Y
BUILD(<i>index, components, filter</i>)	N
BYTES(<i>file</i>)	Y
CLOSE(<i>file</i>)	Y
COPY(<i>file, new file</i>)	Y
CREATE(<i>file</i>)	Y
DUPLICATE(<i>file</i>)	Y
DUPLICATE(<i>key</i>)	Y
EMPTY(<i>file</i>)	Y
EOF(<i>file</i>)	Y ₂
FLUSH(<i>file</i>)	Y
LOCK(<i>file</i>)	Y

NAME(<i>label</i>)	Y
OPEN(<i>file, access mode</i>)	Y
PACK(<i>file</i>)	Y
POINTER(<i>file</i>)	Y ₃
POINTER(<i>key</i>)	Y ₃
POSITION(<i>file</i>)	Y ₄
POSITION(<i>key</i>)	Y ₄
RECORDS(<i>file</i>)	Y
RECORDS(<i>key</i>)	Y
REMOVE(<i>file</i>)	Y
RENAME(<i>file, new file</i>)	Y
SEND(<i>file, message</i>)	Y
SHARE(<i>file, access mode</i>)	Y
STATUS(<i>file</i>)	Y
STREAM(<i>file</i>)	Y
UNLOCK(<i>file</i>)	Y
Record Access	Supported
ADD(<i>file</i>)	Y ₁₀
ADD(<i>file, length</i>)	N
APPEND(<i>file</i>)	Y
APPEND(<i>file, length</i>)	N
DELETE(<i>file</i>)	Y
GET(<i>file, key</i>)	Y
GET(<i>file, filepointer</i>)	Y
GET(<i>file, filepointer, length</i>)	N
GET(<i>key, keypointer</i>)	Y
HOLD(<i>file</i>)	Y

NEXT(<i>file</i>)	Y
NOMEMO(<i>file</i>)	Y
PREVIOUS(<i>file</i>)	Y
PUT(<i>file</i>)	Y ₁₀
PUT(<i>file</i> , <i>filepointer</i>)	Y
PUT(<i>file</i> , <i>filepointer</i> , <i>length</i>)	N
RELEASE(<i>file</i>)	Y
REGET(<i>file</i> , <i>string</i>)	Y
REGET(<i>key</i> , <i>string</i>)	Y
RESET(<i>file</i> , <i>string</i>)	Y
RESET(<i>key</i> , <i>string</i>)	Y
SET(<i>file</i>)	Y
SET(<i>file</i> , <i>key</i>)	Y
SET(<i>file</i> , <i>filepointer</i>)	Y
SET(<i>key</i>)	Y
SET(<i>key</i> , <i>key</i>)	Y
SET(<i>key</i> , <i>keypointer</i>)	Y
SET(<i>key</i> , <i>key</i> , <i>filepointer</i>)	Y
SKIP(<i>file</i> , <i>count</i>)	Y
WATCH(<i>file</i>)	Y
Transaction Processing	Supported
LOGOUT(<i>timeout</i> , <i>file</i> , ..., <i>file</i>)	Y _{6,7}
COMMIT	Y
ROLLBACK	Y

Null Data Processing	Supported
NULL(<i>field</i>)	N
SETNULL(<i>field</i>)	N
SETNONNULL(<i>field</i>)	N

Notes

- 1 We recommend using a variable password that is lengthy and contains special characters because this more effectively hides the password value from anyone looking for it. For example, a password like "dd...#\$...*&" is much more difficult to "find" than a password like "SALARY."

Tip

To specify a variable instead of the actual password in the Owner Name field of the File Properties dialog, type an exclamation point (!) followed by the variable name. For example: !MyPassword.

- 2 These functions are supported but not recommended due to the lack of support in other file systems. NEXT and PREVIOUS post Error 33 if you attempt to read beyond the end of the file.
- 3 POINTER() returns a record number.
- 4 POSITION(file) returns a STRING(4). POSITION(key) returns a STRING the size of the key fields + 4 bytes.
- 5 The RECOVER switch must be "armed" at the beginning of your program in order to support transaction processing. See Driver Strings for more information on the RECOVER function.
- 6 LOGOUT has the effect of LOCKing the file. See also PROP:Logout in the *Language Reference*.
- 7 You cannot LOGOUT an aliased file and its primary file at the same time.
- 8 THREADED files consume additional file handles for each thread that accesses the file.
- 9 OEM conversion is not applied to BINARY MEMOs. The driver assumes BINARY MEMOs are zero padded; otherwise, space padded.

- 10 Prior to Clarion 2.003 in 16-bit programs under Microsoft operating systems, writes (ADD, PUT) did not correctly flush operating system buffers. This problem is corrected with Clarion 2.003 and higher, so that writes are slower but safer. To implement the pre 2.003 behavior, use STREAM and FLUSH.

Clarion:Other

Transaction Processing for Clarion Files

When you issue a LOGOUT statement the Clarion file driver creates a transaction file called programname.TR\$. By default this file is created in the same directory as the program. To create the transaction file elsewhere, add a CWC21 section to the WIN.INI file as follows:

```
[CWC21]
CLATMP=path
```

where path is a directory visible to all users. This statement

```
PUTINI('CWC21','CLATMP',path)
```

creates the correct .INI file section.

During a transaction datafile.LOG files are created for each data file edited during the transaction. These LOG files always reside in the same directory as the data file.

If a system crashes while a transaction is active, no user will be able to access the files until a recovery is run on the files. See the RECOVER send command on how to do this.

LOGOUT has the effect of LOCKing the file.

Field Labels

The Clarion driver only supports fully qualified field names (prefix + label) of 16 characters or less. That is, within the Clarion file (*.DAT) header, the driver truncates prefix + label to the first 16 characters. If the first 16 characters are not unique, the truncation results in duplicate field names.

Duplicate field names within the file header can cause problems with Clarion for DOS 2.1 and earlier. In addition, it can cause problems if you import the file definition from the Clarion file (*.DAT), then compile a Clarion application based on the imported file definition containing the duplicate field names.

You can avoid duplicate field name problems by using the NAME attribute (the **External Name** field in the Data Dictionary's **Field Properties** dialog) to provide unique names for fields whose first 16 characters are duplicated. By providing unique names in the NAME attribute, your application can refer to the field by its (long) label, and the Clarion driver uses the unique NAME attribute to resolve conflicts.

Clipper Database Driver

Clipper:Specifications

The Clipper file driver is compatible with Clipper Summer '87 and Clipper 5.0. The default data file extension is *.DBF.

Keys and Indexes exist as separate files from the data file. Keys are dynamic--they automatically update as the data file changes. Indexes are static--they do not automatically update, but instead require the BUILD statement for updating. The default file extension for the index file is *.NTX.

The driver stores records as fixed length. It stores memo fields in a separate file. The memo file name takes the first eight characters of the File Label plus an extension of .DBT.

Files:

- C60CLPXL.LIB** Windows Static Link Library
- C60CLPX.LIB** Windows Export Library
- C60CLPX.DLL** Windows Dynamic Link Library

Tip

As a popular xBase database application development system, Clipper provides a common file format for many installed business applications and their data files. Use the Clipper driver to access these files in their native format.

Clipper:Data Types

The xBase file format stores all data as ASCII strings. You may either specify STRING types with declared pictures for each field, or specify native Clarion data types, which the driver converts automatically.

<u>Clipper data type</u>	<u>Clarion data type</u>	<u>STRING w/ picture</u>
Date	DATE	STRING(@D12)
*Numeric	REAL	STRING(@N-_p.d)
*Logical	BYTE	STRING(1)
Character	STRING	STRING
*Memo	MEMO	MEMO

If your application reads and writes to existing files, a pictured STRING will suffice. However, if your application *creates* a Clipper file, you may require additional information for these Clipper types:

- † To create a numeric field in the Data Dictionary, choose the REAL data type. In the External Name field on the Attributes tab, specify '*NumericFieldName*=N(*Precision*,*DecimalPlaces*)' where *NumericFieldName* is the name of the field, *Precision* is the precision of the field and *DecimalPlaces* is the number of decimal places. With a REAL data type, you cannot access the Character or Places fields in the Field definition, you must specify those attributes with an expression in the External Name Field on the Attributes tab.

For example, if you want to create a field called Number with nine significant digits and two decimal places, enter 'Number=N(9,2) in the External Name field on the Attributes tab of the Field properties in the Data Dictionary.

If you're hand coding a native Clarion data type, add the NAME attribute using the same syntax.

If you're hand coding a STRING with picture, STRING(@N-_9.2), NAME('Number'), where *Number* is the field name.

- † To create a logical field, using the data dictionary, choose the BYTE data type. There are no special steps; however, see the miscellaneous section for tips on reading the data from the field.

If you're hand coding a STRING with picture, add the NAME attribute: STRING(1), NAME('LogFld = L').

- † To create a date field, using the data dictionary, choose the DATE data type, rather than LONG, which you usually use for the TopSpeed or Clarion file formats.
- † MEMO field declarations require the a pointer field in the file's record structure. Declare the pointer field as a STRING(10) or a LONG. This field will be stored in the .DBF file containing the offset of the memo in the .DBT file. The MEMO declaration must have a NAME() attribute naming the pointer field. An example file declaration follows:

```
File FILE, DRIVER('Clipper')
Memo1 MEMO(200),NAME('Notes')
Memo2 MEMO(200),NAME('Text')
Rec RECORD
Mem1Ptr LONG,NAME('Notes')
Mem2Ptr STRING(10),NAME('Text')
END
END
```

Tip

Whenever possible, use the File Import Utility in the Dictionary Editor to define your files.

Clipper:File Specifications/Maximums

File Size: 2,000,000,000 bytes
Records per File: 1,000,000,000
Record Size: 4,000 bytes (Clipper '87)
8,192 bytes (Clipper 5.0)
Field Size
Character: 254 bytes (Clipper '87)
2048 bytes (Clipper 5.0)
Date: 8 bytes
Logical: 1 byte
Numeric: 20 bytes including decimal point
Memo: 65,520 bytes (see note)
Fields per Record:1024
Keys/Indexes per File: No Limit
Key Sizes
Character: 100 bytes
Numeric, Date: 8 bytes
Memo fields per File: Dependent on available memory
Open Files: Operating system dependent

Clipper:Driver Strings

There are switches or "driver strings" you can set to control the way your application creates, reads, and writes files with a specific driver. Driver strings are simply messages or parameters that are sent to the file driver at run-time to control its behavior. See *Common Driver Features--Driver Strings* for an overview of these runtime Database Driver switches and parameters.

Note:

Some driver strings have no effect after the file is open, so no SEND function syntax is listed for those strings. However, the SEND function syntax to return the value of the switch is listed for all driver strings.

The Clipper Driver supports the following Driver Strings:

BUFFERS

```
DRIVER('CLIPPER', '/BUFFERS = n' )
```

```
[ Status" = ] SEND(file, 'BUFFERS [= n ]' )
```

Sets the size of the buffer used to read and write to the file, where the buffer size is (n * 512 bytes). Use the /BUFFERS driver string to increase the buffer size if access is slow. Maximum buffer size is 4,294,967,264. SEND returns the size of the buffer in bytes.

Tip

The default is three buffers of 1024 bytes each. Increasing the number of buffers will not increase performance when a file is shared by multiple users.

RECOVER

```
DRIVER('CLIPPER', '/RECOVER' )
```

```
[ Status" = ] SEND(file, 'RECOVER' )
```

Equivalent to the Xbase RECALL command, which recovers records marked for deletion. When using the Clipper driver, the DELETE statement flags a record as "inactive." The driver does not remove the record until the PACK command is executed.

RECOVER is evaluated each time you open the file if you add the driver string to the data dictionary. When the driver recovers the records previously marked for deletion, you must manually rebuild keys and indexes with the BUILD statement.

IGNORESTATUS

DRIVER('CLIPPER', '/IGNORESTATUS = on | off ')

[Status" =] SEND(file, 'IGNORESTATUS [on | off] ')

When set *on*, the driver does *not* skip deleted records when accessing the file with GET, NEXT, and PREVIOUS in file order. It also enables a PUT on a deleted or held record. IGNORESTATUS requires opening the file in exclusive mode. SEND returns the IGNORESTATUS setting (ON or OFF) in the form of a STRING(3).

DELETED

[Status" =] SEND(file, 'DELETED')

For use only with the SEND command, when IGNORESTATUS is on. Returns the status of the current record. If deleted, the return string is "ON" and if not, "OFF."

ZEROY2K

DRIVER('CLIPPER', '/ZEROY2K = on | off')

[Status" =] SEND(file, 'ZEROY2K [on | off] ')

In the header of Clipper files there is a field that stores the year that the file was last edited. Some applications store this as the number of years since 1900. Others store it as a 2-digit year. So for dates in the year 2000 some applications store 0 in this field and others 100. Clarion will read files with either. However it will write 100. Writing 100 may make the files unreadable by products that only support 0. To change this behavior you can with use a driver string of ZEROY2K, a SEND command or a setting in the WIN.INI file.

The driver will store 0 in the DBF file header when the WINI.INI setting is set to 1 or 'on' in a SEND command or driver string, otherwise a 100 will be stored in the DBF file header.

Note: The SEND command causes the setting to be set for all files that use that driver, not just for that file.

Example:

```

WIN.INI
;Sets all Clipper files to store a 0 in the DBF file header
[CWCLIPPER]
ZEROY2K=1
SEND command
SEND('Orders', ZEROY2K='on' !sets Orders file to store 0 in the DBF file header
SEND('Orders', ZEROY2K='off' !sets Orders file to store 100 in the DBF file header
Driver String
Orders FILE, DRIVER('clipper', '/ZEROY2K=on'),PRE(ORD) !SETS Orders file to store 0

```

Clipper:Supported Commands and Attributes

File Attributes	Supported
CREATE	Y ₁
DRIVER(<i>filetype</i> [, <i>driver string</i>])	Y
NAME	Y
ENCRYPT	N
OWNER(<i>password</i>)	N
RECLAIM	N ₂
PRE(<i>prefix</i>)	Y
BINDABLE	Y
THREAD	Y ₁₆
EXTERNAL(<i>member</i>)	Y
DLL(<i>[flag]</i>)	Y
OEM	Y
File Structures	Supported
INDEX	Y
KEY	Y
MEMO	Y ₃
BLOB	N
RECORD	Y

Index, Key, Memo Attributes Supported

BINARY	N
DUP	Y ₄
NOCASE	Y
OPT	N
PRIMARY	Y
NAME	Y ₃
Ascending Components	Y
Descending Components	Y
Mixed Components	Y

Field Attributes Supported

DIM	N
OVER	Y
NAME	Y ₁

File Procedures Supported

BOF(<i>file</i>)	Y ₁₁
BUFFER(<i>file</i>)	N
BUILD(<i>file</i>)	Y
BUILD(<i>key</i>)	Y
BUILD(<i>index</i>)	Y
BUILD(<i>index, components</i>)	Y ₅
BUILD(<i>index, components, filter</i>)	N
BYTES(<i>file</i>)	N
CLOSE(<i>file</i>)	Y
COPY(<i>file, new file</i>)	Y ₆
CREATE(<i>file</i>)	Y ₁
DUPLICATE(<i>file</i>)	Y

DUPLICATE(<i>key</i>)	Y
EMPTY(<i>file</i>)	Y
EOF(<i>file</i>)	Y ₁₁
FLUSH(<i>file</i>)	Y
LOCK(<i>file</i>)	Y
NAME(<i>label</i>)	Y
OPEN(<i>file, access mode</i>)	Y ₇
PACK(<i>file</i>)	Y
POINTER(<i>file</i>)	Y ₁₂
POINTER(<i>key</i>)	Y ₁₂
POSITION(<i>file</i>)	Y ₁₃
POSITION(<i>key</i>)	Y ₁₃
RECORDS(<i>file</i>)	Y ₁₄
RECORDS(<i>key</i>)	Y ₁₄
REMOVE(<i>file</i>)	Y
RENAME(<i>file, new file</i>)	Y ₈
SEND(<i>file, message</i>)	Y
SHARE(<i>file, access mode</i>)	Y ₇
STATUS(<i>file</i>)	Y
STREAM(<i>file</i>)	Y
UNLOCK(<i>file</i>)	Y
Record Access	Supported
ADD(<i>file</i>)	Y ₉
ADD(<i>file, length</i>)	N
APPEND(<i>file</i>)	Y ₉
APPEND(<i>file, length</i>)	N
DELETE(<i>file</i>)	Y ₆

GET(<i>file,key</i>)	Y
GET(<i>file, filepointer</i>)	Y
GET(<i>file, filepointer, length</i>)	N
GET(<i>key, keypointer</i>)	Y
HOLD(<i>file</i>)	Y ¹⁰
NEXT(<i>file</i>)	Y
NOMEMO(<i>file</i>)	Y
PREVIOUS(<i>file</i>)	Y
PUT(<i>file</i>)	Y
PUT(<i>file, filepointer</i>)	Y
PUT(<i>file, filepointer, length</i>)	N
RELEASE(<i>file</i>)	Y
REGET(<i>file,string</i>)	Y
REGET(<i>key,string</i>)	Y
RESET(<i>file,string</i>)	Y
RESET(<i>key,string</i>)	Y
SET(<i>file</i>)	Y
SET(<i>file, key</i>)	Y
SET(<i>file, filepointer</i>)	Y
SET(<i>key</i>)	Y
SET(<i>key, key</i>)	Y
SET(<i>key, keypointer</i>)	Y
SET(<i>key, key, filepointer</i>)	Y
SKIP(<i>file, count</i>)	Y
WATCH(<i>file</i>)	Y
Transaction Processing	Supported (See Note 15)
LOGOUT(<i>timeout, file, ..., file</i>)	N

COMMIT	N
ROLLBACK	N
Null Data Processing	Supported
NULL(<i>field</i>)	N
SETNULL(<i>field</i>)	N
SETNONNULL(<i>field</i>)	N

Notes

- 1 If your application *creates* a Clipper file, you may require additional NAME information for these Clipper data types:

For a Clipper numeric field, use the Clarion REAL data type. Then in the NAME attribute (the **External Name** field on the **Attributes** tab in the **Field Properties** dialog), specify '*NumericFieldName*=N(*Precision*,*DecimalPlaces*)' where *NumericFieldName* is the name of the field, *Precision* is the precision of the field and *DecimalPlaces* is the number of decimal places. See *Data Types* above for more information.

For a Clipper logical field, use the Clarion BYTE data type. See *Data Types* above for more information. See the *Miscellaneous* section for tips on reading the data from the field.

For a Clipper date field, use the Clarion DATE data type. See *Data Types* above for more information.

- 2 When the driver deletes a record from a Clipper database, the record is not physically removed, instead the driver marks it inactive. Memo fields are not physically removed from the memo file, however they cannot be retrieved if they refer to an inactive record. To remove records and memo fields permanently, execute a PACK(file).

Tip

To those programmers familiar with Clipper, this driver processes deleted records consistent with the way Clipper processes them after the SET DELETED ON command is issued. Records marked for deletion are ignored from processing by executable code statements, but remain in the data file.

- 3 MEMO field declarations require a pointer field in the file's record structure. Declare the pointer field as a STRING(10) or a LONG. This field will be stored in the .DBF file containing the offset of the memo in the .DBT file. The MEMO declaration must have a NAME() attribute naming the pointer field. See *Data Types* above for more information.

- 4 In Clipper it is legal to enter multiple records with duplicates of the unique key components. However, only the first of these records is indexed. So processing in key order only shows this first record. If you delete a record, then enter a new record with the same key value, the key file continues to point at the deleted record rather than the new record. In this situation, the Clipper file driver driver changes the key file to point at the active record rather than the deleted record. This means that if you use a Clipper program to delete a unique record, then insert a duplicate of this record, the new record is invisible when processing in key order until a pack is done. If you do the same process in a Clarion program, the new record is visible when processing in key order.

- 5 When building dynamic indexes, the *components* may take one of two forms:

```
BUILD(DynNdx, '+Pre:FLD1, -Pre:FLD2')
```

This form specifies the names of the fields on which to build the index. The field names must appear as specified in the fields' NAME() attribute if supplied, or must be the label name. A prefix may be used for compatibility with Clarion conventions but is ignored.

```
BUILD(DynNdx, 'T[Expression]')
```

This form specifies the type and Expression used to build an index--see *Miscellaneous--Key Definition* below.

- 6 The COPY() command copies data and memo files using *newfile*, which may specify a new file name or directory. Key or index files are copied if the *newfile* is a subdirectory specification. To copy an index file to a new file, use a special form of the copy command:

```
COPY(file, '<index>|<newfile>')
```

This returns *File Not Found* if an invalid index is passed. The COPY command assumes a default extension of .NTX for both the source and the target file names if none is specified. If you require a file name without an extension, terminate the name with a period. Given the file structure:

```
Clar2  FILE,CREATE,DRIVER('Clipper'),PRE(CL2)
NumKey  KEY(+CL2:Num),DUP
StrKey  KEY(+CL2:Str1)
StrKey2 KEY(+CL2:Str2)
Amemo   MEMO(100),NAME('mem')
Record  RECORD
Num     STRING(@n-_9.2)
STR1    STRING(2)
STR2    STRING(2)
Mem     STRING(10)
        END
        END
```

The following commands copy this file definition to A:

```
COPY(Clar2, 'A:\CLAR2')  
COPY(Clar2, 'StrKey|A:\STRKEY')  
COPY(Clar2, 'StrKey2|A:\STRKEY2')  
COPY(Clar2, 'NumKey|A:\NUMKEY')
```

After these calls, the following files would exist on drive A: CLAR2.DBF, CLAR2.DBT, STRKEY.NTX, STRKEY2.NTX, and NUMKEY.NTX.

- 7 You do not need SHARE (or VSHARE) in any environment (for example, Novell) that supplies file locking as part of the operating system.
- 8 The RENAME command copies the data and memo files using *newfile*, which may specify a new file name or directory path. Key and index files must be renamed using the same syntax as the COPY command, above.
- 9 The ADD statement tests for duplicate keys before modifying the data file or its associated KEY files. Consequently it is slower than APPEND which performs no checks and does not update KEYS. When adding large amounts of data to a database use APPEND...BUILD in preference to ADD.
- 10 Clipper performs record locking by locking the entire record within the data file. This prevents read access to other processes. Therefore we recommend minimizing the amount of time for which a record is held.
- 11 Although the driver supports these functions, we do not recommend their use. They must physically access the files and add overhead. Instead, test the value returned by ERRORCODE() after each sequential access. NEXT or PREVIOUS post Error 33 (Record Not Available) if an attempt is made to access a record beyond the end or beginning of the file.
- 12 There is no distinction between file pointers and key pointers; they both return the record number for any given record.
- 13 POSITION(file) returns a STRING(12). POSITION(key) returns a STRING the size of the key fields + 4 bytes.
- 14 Under Clipper, the RECORDS() function reports the same number of records for the data file and its keys and indexes. Usually there will be no difference in the number of records *unless* the INDEX is out of date. Because the DELETE statement does not physically remove records, *the number of records reported by the RECORDS() function includes inactive records*. Exercise care when using this function.
- 15 See also PROP:Logout in the *Language Reference*.
- 16 THREADED files consume additional file handles for each thread that accesses the file.

Clipper:Other

Boolean Evaluation

- † Clipper allows a logical field to accept one of nine possible values (y,Y,n,N,t,T,f,F or a space character). The space character is neither true nor false. When using a logical field from a preexisting database in a logical expression, account for all these possibilities. Remember that when a STRING field is used as an expression, it is true if it contains any data and false if it is equal to zero or blank. Therefore, to evaluate a Logical field's truth, the expression should be true if the field contains any of the "true" characters (T,t,Y, or y). For example, if a Logical field were used to specify a product as taxable or nontaxable, the expression to evaluate its truth would be:

(If Condition):

```
Taxable='T' OR Taxable='t' OR Taxable='Y' OR Taxable='y'
```

Large MEMOs

- † Clarion supports MEMO fields up to a maximum of 64K. If you have an existing file which includes a memo greater than 64K, you can use the file but not modify the large MEMOs.
- † You can determine when your application encounters a large MEMO by detecting when the memo pointer variable is non-blank, but the memo appears to be blank. Error 47 (Bad Record Declaration) is posted. If you attempt to update such a record, any modification to the MEMO field is ignored.

Long Field Names

- † Clipper supports a maximum of 10 characters in a field name. If you require more, use an External Name with 10 characters or less.

Sort Sequence

- † The Clipper driver supports international sort orders, however, to maintain compatibility with Clipper's international sort order, remove the CLADIGRAPH= line from ..\C60 root)\BIN\Clarion6.ENV file.

Key Definition

- † Clipper supports the use of expressions to define keys. Within the Dictionary Editor, you can place the expression in the external name field in the **Key Properties** dialog. The format of the external name is:

```
'FileName=T[Expression]'
```

Where **FileName** represents the name of the index file (which can contain a path and file extension), and τ represents the type of the index. Valid types are: C = character, D = date, and N = numeric. If the type is D or N then **Expression** can name only one field.

String expressions may use the '+' operator to concatenate multiple string arguments. Numeric expressions use the '+' or '-' operators with their conventional meanings. The maximum length of a Clipper expression is 250 characters.

The expression may refer to multiple fields in the record, and may contain xBase functions. Square brackets must enclose the expression. The currently supported functions appear below. If the driver encounters an unsupported Xbase function in a preexisting file, it posts error 76 'Invalid Index String' when the file is opened for keys and static indexes.

Supported xBase Key Definition Functions

ALLTRIM(string)	Removes leading and trailing spaces.
CTOD(string)	Converts a string key to a date. The <i>string</i> must be in the format mm/dd/yy; the result takes the form 'yyyymmdd'. The yyyy element of the date defaults to the twentieth century. An invalid date results in a key containing blanks.
DELETED()	Returns TRUE if the record is deleted.
DESCEND(string date numeric)	Inverts the argument, and creates descending Clipper indexes.
DTOC(date)	Converts a date key to string format 'mm/dd/yy'
DTOS(date)	Converts a date key to string format 'yyyymmdd'
FIXED(float)	Converts a float key to a numeric.
FLOAT(numeric)	Converts a numeric key to a float.
IIF(bool, val1, val2)	Returns val1 if the first parameter is TRUE, otherwise returns val2.
LEFT(string, n)	Returns the leftmost <i>n</i> characters of the string key as a string of length <i>n</i> .
LOWER(string)	Converts a string key to lower case.
LTRIM(string)	Removes spaces from the left of a string.
RECNO()	Returns the current record number.
RIGHT(string, n)	Returns the rightmost <i>n</i> characters of the string key as a string of length <i>n</i> .
RTRIM(string)	Removes spaces from the right of a string.

<code>STR(numeric [,length[, decimal places]])</code>	Converts a numeric to a string. The length of the string and the number of decimal places are optional. The default string length is 10, and the number of decimal places is 0.
<code>SUBSTR(string,offset,n)</code>	Returns a substring of the <i>string</i> key starting at <i>offset</i> and of <i>n</i> characters in length.
<code>TRIM(string)</code>	Removes spaces from the right of a string (identical to RTRIM).
<code>UPPER(string)</code>	Converts a string key to upper case.
<code>VAL(string)</code>	Converts a string key to a numeric.

dBaselll Database Driver

dBaselll:Specifications

The dBase3 file driver is compatible with dBase III. The default data file extension is *.DBF.

Keys and Indexes exist as separate files from the data file. Keys are dynamic--they automatically update as the data file changes. Indexes are static--they do not automatically update, but instead require the BUILD statement for updating. The default file extension for the index file is *.NDX. International sort orders are supported.

The driver stores records as fixed length. It stores memo fields in a separate file. The memo file name takes the first eight characters of the File Label plus an extension of .DBT.

Files:	C60DB3XL.LIB	Windows Static Link Library
	C60DB3X.LIB	Windows Export Library
	C60DB3X.DLL	Windows Dynamic Link Library

Tip

dBase III is probably the most common file format for PC database applications. These days, even desktop publishing programs can import dBase III compatible .DBF files. If the main task of your application is to export data files for other applications about which you know nothing, you should consider this format.

dBaselll:Data Types

The xBase file format stores all data as ASCII strings. You may either specify STRING types with declared pictures for each field, or specify native Clarion types, which the driver converts automatically.

<u>dBase data type</u>	<u>Clarion data type</u>	<u>STRING w/ picture</u>
Date	DATE	STRING(@D12)
*Numeric	REAL	STRING(@N- <u>p</u> .d)
*Logical	BYTE	STRING(1)
Character	STRING	STRING
*Memo	MEMO	MEMO

If your application reads and writes to existing files, a pictured STRING will suffice. However, if your application *creates* a dBase III file, you may require additional information for these dBase III types:

- † To create a numeric field in the Data Dictionary, choose the REAL data type. In the External Name field on the Attributes tab, specify '*NumericFieldName*=N(*Precision*,*DecimalPlaces*)' where *NumericFieldName* is the name of the field, *Precision* is the precision of the field and *DecimalPlaces* is the number of decimal places. With a REAL data type, you cannot access the Character or Places fields in the Field definition, you must specify those attributes with an expression in the External Name Field on the Attributes tab.

For example, if you want to create a field called Number with nine significant digits and two decimal places, enter 'Number=N(9,2) in the External Name field on the Attributes tab of the Field properties in the Data Dictionary.

If you're hand coding a native Clarion data type, add the NAME attribute using the same syntax.

If you're hand coding a STRING with picture, STRING(@N-9.2), NAME('Number'), where *Number* is the field name.

To create a logical field, using the data dictionary, choose the BYTE data type. There are no special steps; however, see the miscellaneous section for tips on reading the data from the field.

If you're hand coding a STRING with picture, add the NAME attribute: STRING(1), NAME('LogFld = L').

To create a date field, using the data dictionary, choose the DATE data type, rather than LONG, which you usually use for the TopSpeed or Clarion file formats.

MEMO field declarations require the a pointer field in the file's record structure. Declare the pointer field as a STRING(10) or a LONG. This field will be stored in the .DBF file containing the offset of the memo in the .DBT file. The MEMO declaration must have a NAME() attribute naming the pointer field. An example file declaration follows:

```
File FILE, DRIVER('dBase3')
Memo1 MEMO(200),NAME('Notes')
Memo2 MEMO(200),NAME('Text')
Rec RECORD
Mem1Ptr LONG,NAME('Notes')
Mem2Ptr STRING(10),NAME('Text')
      END
    END
```

Tip

Use the File Import Utility in the Clarion Dictionary Editor to define your files.

dBaseIII:File Specifications/Maximums

File Size: 2,000,000,000 bytes
Records per File: 1,000,000,000
Record Size: 4,000 bytes
Field Size
Character: 254 bytes
Date: 8 bytes
Logical: 1 byte
Numeric: 20 bytes including decimal point
Memo: 64K (see note)
Fields per Record: 128
Keys/Indexes per File: No Limit
Key Sizes
Character: 100 bytes
Numeric, Date: 8 bytes
Memo fields per File: Dependent on available memory
Open Files: Operating system dependent

dBaseIII:Driver Strings

There are switches or "driver strings" you can set to control the way your application creates, reads, and writes files with a specific driver. Driver strings are simply messages or parameters that are sent to the file driver at run-time to control its behavior. See *Common Driver Features--Driver Strings* for an overview of these runtime Database Driver switches and parameters.

Note:

Some driver strings have no effect after the file is open, so no SEND function syntax is listed for those strings. However, the SEND function syntax to return the value of the switch is listed for all driver strings.

The dBaseIII Driver supports the following Driver Strings:

BUFFERS

DRIVER('DBASE3', '/BUFFERS = n')

[Status" =] SEND(file, 'BUFFERS [= n]')

Sets the size of the buffer used to read and write to the file, where the buffer size is (n * 512 bytes). Use the /BUFFERS driver string to increase the buffer size if access is slow. Maximum buffer size is 4,294,967,264. SEND returns the size of the buffer in bytes.

Tip

The default is three buffers of 1024 bytes each. Increasing the number of buffers will not increase performance when a file is shared by multiple users.

RECOVER

DRIVER('DBASE3', '/RECOVER')

[Status" =] SEND(file, 'RECOVER')

Equivalent to the Xbase RECALL command, which recovers records marked for deletion. When using the dBaseIV driver, the DELETE statement flags a record as "inactive." The driver does not remove the record until the PACK command is executed.

RECOVER is evaluated each time you open the file if you add the driver string to the data dictionary. When the driver recovers the records previously marked for deletion, you must manually rebuild keys and indexes with the BUILD statement.

IGNORESTATUS

```
DRIVER('DBASE3', '/IGNORESTATUS = on | off ' )
```

```
[ Status" = ] SEND(file, 'IGNORESTATUS [ on | off ] ' )
```

When set *on*, the driver does *not* skip deleted records when accessing the file with GET, NEXT, and PREVIOUS in file order. It also enables a PUT on a deleted or held record. IGNORESTATUS requires opening the file in exclusive mode. SEND returns the IGNORESTATUS setting (ON or OFF) in the form of a STRING(3).

DELETED

```
[ Status" = ] SEND(file, 'DELETED' )
```

For use only with the SEND command, when IGNORESTATUS is on. Returns the status of the current record. If deleted, the return string is "ON" and if not, "OFF."

OMNIS

```
DRIVER('DBASE3', '/OMNIS' )
```

```
SEND(file, 'OMNIS' )
```

Specifies OMNIS file header and file delimiter compatibility. SEND is only valid when the file is closed; it returns nothing.

ZEROY2K

```
DRIVER('DBASE3', '/ZEROY2K = on | off' )
```

```
[ Status" = ] SEND(file, 'ZEROY2K [ on | off ] ' )
```

In the header of dBase3files there is a field that stores the year that the file was last edited. Some applications store this as the number of years since 1900. Others store it as a 2 digit year. So for dates in the year 2000 some applications store 0 in this field and others 100. Clarion will read files with either. However it will write 100. Writing 100 may make the files unreadable by products that only support 0. To change this behavior you can with use a driver string of ZEROY2K, a SEND command or a setting in the WIN.INI file.

The driver will store 0 in the DBF file header when the WINI.INI setting is set to 1 or 'on' in a SEND command or driver string, otherwise a 100 will be stored in the DBF file header.

Note: The SEND command causes the setting to be set for all files that use that driver, not just for that file.

Example:

```
WIN.INI
;Sets all dBase3 files to store a 0 in the DBF file header
[CWDBASE3]
ZEROY2K=1
!SEND command
SEND('Orders', ZEROY2K='on' !sets Orders file to store 0 in the DBF file header
SEND('Orders', ZEROY2K='off' !sets Orders file to store 100 in the DBF file header
!Driver String Orders
FILE, DRIVER('dbase3', '/ZEROY2K=on'),PRE(ORD) !SETS Orders file to store 030 -
```

File Attributes Supported

CREATE	Y
DRIVER(<i>filetype</i> [, <i>driver string</i>])	Y
NAME	Y
ENCRYPT	N
OWNER(<i>password</i>)	N
RECLAIM	N ₁
PRE(<i>prefix</i>)	Y
BINDABLE	Y
THREAD	Y ₁₂
EXTERNAL(<i>member</i>)	Y
DLL(<i>[flag]</i>)	Y
OEM	Y

File Structures Supported

INDEX	Y
KEY	Y
MEMO	Y
BLOB	N
RECORD	Y

Index, Key, Memo Attributes Supported

BINARY	N
DUP	Y ₂
NOCASE	Y
OPT	N
PRIMARY	Y
NAME	Y
Ascending Components	Y
Descending Components	Y
Mixed Components	N
Field Attributes	Supported
DIM	N
OVER	Y
NAME	Y
File Procedures	Supported
BOF(<i>file</i>)	Y ₈
BUFFER(<i>file</i>)	N
BUILD(<i>file</i>)	Y
BUILD(<i>key</i>)	Y
BUILD(<i>index</i>)	Y
BUILD(<i>index, components</i>)	Y ₃
BUILD(<i>index, components, filter</i>)	N
BYTES(<i>file</i>)	N
CLOSE(<i>file</i>)	Y
COPY(<i>file, new file</i>)	Y ₄
CREATE(<i>file</i>)	Y
DUPLICATE(<i>file</i>)	Y

DUPLICATE(<i>key</i>)	Y
EMPTY(<i>file</i>)	Y
EOF(<i>file</i>)	Y ₈
FLUSH(<i>file</i>)	Y
LOCK(<i>file</i>)	N
NAME(<i>label</i>)	Y
OPEN(<i>file, access mode</i>)	Y ₅
PACK(<i>file</i>)	Y
POINTER(<i>file</i>)	Y ₉
POINTER(<i>key</i>)	Y ₉
POSITION(<i>file</i>)	Y ₁₀
POSITION(<i>key</i>)	Y ₁₀
RECORDS(<i>file</i>)	Y ₁₁
RECORDS(<i>key</i>)	Y ₁₁
REMOVE(<i>file</i>)	Y
RENAME(<i>file, new file</i>)	Y ₄
SEND(<i>file, message</i>)	Y
SHARE(<i>file, access mode</i>)	Y ₅
STATUS(<i>file</i>)	Y
STREAM(<i>file</i>)	Y
UNLOCK(<i>file</i>)	N
Record Access	Supported
ADD(<i>file</i>)	Y ₆
ADD(<i>file, length</i>)	N
APPEND(<i>file</i>)	Y ₆
APPEND(<i>file, length</i>)	N
DELETE(<i>file</i>)	Y ₁

GET(<i>file,key</i>)	Y
GET(<i>file, filepointer</i>)	Y
GET(<i>file, filepointer, length</i>)	N
GET(<i>key, keypointer</i>)	Y
HOLD(<i>file</i>)	Y7
NEXT(<i>file</i>)	Y
NOMEMO(<i>file</i>)	Y
PREVIOUS(<i>file</i>)	Y
PUT(<i>file</i>)	Y
PUT(<i>file, filepointer</i>)	Y
PUT(<i>file, filepointer, length</i>)	N
RELEASE(<i>file</i>)	Y
REGET(<i>file,string</i>)	Y
REGET(<i>key,string</i>)	Y
RESET(<i>file,string</i>)	Y
RESET(<i>key,string</i>)	Y
SET(<i>file</i>)	Y
SET(<i>file, key</i>)	Y
SET(<i>file, filepointer</i>)	Y
SET(<i>key</i>)	Y
SET(<i>key, key</i>)	Y
SET(<i>key, keypointer</i>)	Y
SET(<i>key, key, filepointer</i>)	Y
SKIP(<i>file, count</i>)	Y
WATCH(<i>file</i>)	Y
Transaction Processing	Supported
LOGOUT(<i>timeout, file, ..., file</i>)	N

COMMIT	N
--------	---

ROLLBACK	N
----------	---

Null Data Processing Supported

NULL(<i>field</i>)	N
----------------------	---

SETNULL(<i>field</i>)	N
-------------------------	---

SETNONNULL(<i>field</i>)	N
----------------------------	---

Notes

- 1 When the driver deletes a record from a dBase III database, the record is not physically removed, instead the driver marks it inactive. Memo fields are not physically removed from the memo file, however they cannot be retrieved if they refer to an inactive record. Key values *are* removed from the index files. To remove records and memo fields permanently, execute a PACK(file).

Tip

To those programmers familiar with dBase III, this driver processes deleted records consistent with the way dBase III processes them after the SET DELETED ON command is issued. Records marked for deletion are ignored from processing by executable code statements, but remain in the data file.

- 2 dBase III does not support any form of unique index. So the DUP attribute must be on all keys.
- 3 When building dynamic indexes, the *components* may take one of two forms:

```
BUILD(DynNdx, '+Pre:FLD1, -Pre:FLD2')
```

This form specifies the names of the fields on which to build the index. The field names must appear as specified in the fields' NAME() attribute if supplied, or must be the label name. A prefix may be used for compatibility with Clarion conventions but is ignored.

```
BUILD(DynNdx, 'T[Expression]')
```

This form specifies the type and Expression used to build an index--see *Miscellaneous--Key Definition* below.

- 4 These commands copy data and memo files using *newfile*, which may specify a new file name or directory. Key or index files are copied if the *newfile* is a subdirectory specification. To copy an index file to a new file, use a special form of the copy or rename command:

```
COPY(file, '<index>|<newfile>')
```

This returns *File Not Found* if an invalid index is passed. The COPY command assumes a default extension of ".NDX" for both the source and the target file names if none is specified. If you require a file name without an extension, terminate the name with a period. Given the file structure:

```
Clar2 FILE,CREATE,DRIVER('dBase3'),PRE(CL2)
NumKey KEY(+CL2:Num),DUP
StrKey KEY(+CL2:Str1)
StrKey2 KEY(+CL2:Str2)
AMemo MEMO(100),NAME('mem')
Record RECORD
Num STRING(@n-9.2)
STR1 STRING(2)
STR2 STRING(2)
Mem STRING(10)
END
END
```

The following commands copy this file definition to A:

```
COPY(Clar2, 'A:\CLAR2')
COPY(Clar2, 'StrKey|A:\STRKEY')
COPY(Clar2, 'StrKey2|A:\STRKEY2')
COPY(Clar2, 'NumKey|A:\NUMKEY')
```

After these calls, the following file would exist on drive A: CLAR2.DBF, CLAR2.DBT, STRKEY.NDX, STRKEY2.NDX, and NUMKEY.NDX.

- 5 You do not need SHARE (or VSHARE) in any environment (for example, Novell) that supplies file locking as part of the operating system.
- 6 The ADD statement tests for duplicate keys before modifying the data file or its associated KEY files. Consequently it is slower than APPEND which performs no checks and does not update KEYS. When adding large amounts of data to a database use APPEND...BUILD in preference to ADD.
- 7 dBase III performs record locking by locking the entire record within the data file. This prevents read access to other processes. Therefore we recommend minimizing the amount of time for which a record is held.

- 8 Although the driver supports these functions, we do not recommend their use. They must physically access the files and add overhead. Instead, test the value returned by `ERRORCODE()` after each sequential access. `NEXT` or `PREVIOUS` post Error 33 (Record Not Available) if an attempt is made to access a record beyond the end or beginning of the file.
- 9 There is no distinction between file pointers and key pointers; they both return the record number for any given record.
- 10 `POSITION(file)` returns a `STRING(12)`. `POSITION(key)` returns a `STRING` the size of the key fields + 4 bytes.
- 11 Under dBase III, the `RECORDS()` function reports the same number of records for the data file and its keys and indexes. Usually there will be no difference in the number of records *unless* the `INDEX` is out of date. Because the `DELETE` statement does not physically remove records, *the number of records reported by the `RECORDS()` function includes inactive records*. Exercise care when using this function.
- 12 `THREADED` files consume additional file handles for each thread that accesses the file.

dBaseIII:Other

Boolean Evaluation

- † dBase III allows a logical field to accept one of nine possible values (y,Y,n,N,t,T,f,F or a space character). The space character is neither true nor false. When using a logical field from a preexisting database in a logical expression, account for all these possibilities. Remember that when a STRING field is used as an expression, it is true if it contains any data and false if it is equal to zero or blank. Therefore, to evaluate a Logical field's truth, the expression should be true if the field contains any of the "true" characters (T,t,Y, or y). For example, if a Logical field were used to specify a product as taxable or nontaxable, the expression to evaluate its truth would be:

(If Condition):

```
Taxable='T' OR Taxable='t' OR Taxable='Y' OR Taxable='y'
```

Large MEMOs

- † Clarion supports MEMO fields up to a maximum of 64K. If you have an existing file which includes a memo greater than 64K, you can use the file but not modify the large MEMOs.
- † You can determine when your application encounters a large MEMO by detecting when the memo pointer variable is non-blank, but the memo appears to be blank. Error 47 (Bad Record Declaration) is posted, and any modification to the MEMO field is ignored.

Long Field Names

- † dBase III supports a maximum of 10 characters in a field name. If you require more, use an External Name with 10 characters or less.

International Sort Sequence

- † The dBaseIII driver supports international sort orders, however, to maintain compatibility with dBaseIII's international sort order, remove the CLADIGRAPH= line from ..\BIN\CLARION6.ENV file.

KEY Definitions

- † dBase III supports the use of expressions to define keys. Within the Dictionary Editor, you can place the expression in the external name field in the *Key Properties* dialog. The general format of the external name is :

'FileName=T[Expression]'

Where **FileName** represents the name of the index file (which can contain a path and file extension), and **T** represents the type of the index. Valid types are: C = character, D = date, and N = numeric. If the type is D or N then **Expression** can name only one field.

String expressions may use the '+' operator to concatenate multiple string arguments. Numeric expressions use the '+' or '-' operators with their conventional meanings. The maximum length of a dBase III expression is 250 characters.

The expression may refer to multiple fields in the record, and contain xBase functions. Square brackets must enclose the expression. The currently supported functions appear below. If the driver encounters an unsupported Xbase function in a preexisting file, it posts error 76 'Invalid Index String' when the file is opened for keys and static indexes.

Supported xBase Key Definition Functions

ALLTRIM(string)	Removes leading and trailing spaces.
CTOD(string)	Converts a string key to a date. The <i>string</i> must be in the format mm/dd/yy; the result takes the form 'yyyymmdd'. The yyyy element of the date defaults to the twentieth century. An invalid date results in a key containing blanks.
DELETED()	Returns TRUE if the record is deleted.
DTOC(date)	Converts a date key to string format 'mm/dd/yy.'
DTOS(date)	Converts a date key to string format 'yyyymmdd.'
FIXED(float)	Converts a float key to a numeric.
FLOAT(numeric)	Converts a numeric key to a float.
IIF(bool, val1, val2)	Returns val1 if the first parameter is TRUE, otherwise returns val2.
LEFT(string, n)	Returns the leftmost <i>n</i> characters of the string key as a string of length <i>n</i> .
LOWER(string)	Converts a string key to lower case.
LTRIM(string)	Removes spaces from the left of a string.
RECNO()	Returns the current record number.

<code>RIGHT(string, n)</code>	Returns the rightmost <i>n</i> characters of the string key as a string of length <i>n</i> .
<code>RTRIM(string)</code>	Removes spaces from the right of a string.
<code>STR(numeric [,length [, decimal places]])</code>	Converts a numeric to a string. The length of the string and the number of decimal places are optional. The default string length is 10, and the number of decimal places is 0.
<code>SUBSTR(string,offset,n)</code>	Returns a substring of the <i>string</i> key starting at <i>offset</i> and of <i>n</i> characters in length.
<code>TRIM(string)</code>	Removes spaces from the right of a string (identical to RTRIM).
<code>UPPER(string)</code>	Converts a string key to upper case.
<code>VAL(string)</code>	Converts a string key to a numeric.

dBaseIV Database Driver

dBaseIV:Specifications

The dBase4 file driver is compatible with dBase IV. The default data file extension is *.DBF.

Keys and Indexes exist as separate files from the data file. Keys are dynamic--they automatically update as the data file changes. Indexes are static--they do not automatically update, but instead require the BUILD statement for updating. The default file extension for the index file is *.NDX.

dBase IV supports multiple index files, whose extension is *.MDX. The miscellaneous section describes procedures for using .MDX files.

The driver stores records as fixed length. It stores memo fields in a separate file. The memo file name takes the first eight characters of the File Label plus an extension of .DBT.

Files:	C60DB4XL.LIB	Windows Static Link Library
	C60DB4X.LIB	Windows Export Library
	C60DB4X.DLL	Windows Dynamic Link Library

Tip

dBase IV was never as widely adopted as dBase III. Choose this driver only when you must share data with an end-user using dBase IV.

dBaseIV:Data Types

The xBase file format stores all data as ASCII strings. You may either specify STRING types with declared pictures for each field, or specify native Clarion types, which the driver converts automatically.

<u>dBase data type</u>	<u>Clarion data type</u>	<u>STRING w/ picture</u>
Date	DATE	STRING(@D12)
*Numeric	REAL	STRING(@N- <u>p</u> .d)
*Logical	BYTE	STRING(1)
Character	STRING	STRING
*Memo	MEMO	MEMO

If your application reads and writes to existing files, a pictured STRING will suffice. However, if your application *creates* a dBase IV file, you may require additional information for these dBase IV types:

- † To create a numeric field in the Data Dictionary, choose the REAL data type. In the External Name field on the Attributes tab, specify '*NumericFieldName*=N(*Precision*,*DecimalPlaces*)' where *NumericFieldName* is the name of the field, *Precision* is the precision of the field and *DecimalPlaces* is the number of decimal places. With a REAL data type, you cannot access the Character or Places fields in the Field definition, you must specify those attributes with an expression in the External Name Field on the Attributes tab.

For example, if you want to create a field called Number with nine significant digits and two decimal places, enter 'Number=N(9,2) in the External Name field on the Attributes tab of the Field properties in the Data Dictionary.

If you're hand coding a native Clarion data type, add the NAME attribute using the same syntax.

If you're hand coding a STRING with picture, STRING(@N-_9.2), NAME('Number'), where *Number* is the field name.

- † To create a logical field, using the data dictionary, choose the BYTE data type. There are no special steps; however, see the miscellaneous section for tips on reading the data from the field.

If you're hand coding a STRING with picture, add the NAME attribute: STRING(1), NAME('LogFld = L').

- † To create a date field, using the data dictionary, choose the DATE data type, rather than LONG, which you usually use for the TopSpeed or Clarion file formats.
- † MEMO field declarations require the a pointer field in the file's record structure. Declare the pointer field as a STRING(10) or a LONG. This field will be stored in the .DBF file containing the offset of the memo in the .DBT file. The MEMO declaration must have a NAME() attribute naming the pointer field. An example file declaration follows:

```
File FILE, DRIVER('dBase4')
Memo1 MEMO(200),NAME('Notes')
Memo2 MEMO(200),NAME('Text')
Rec RECORD
Mem1Ptr LONG,NAME('Notes')
Mem2Ptr STRING(10),NAME('Text')
END
END
```

Tip

Use the File Import Utility in the Clarion Dictionary Editor to define your files.

dBaseIV:File Specifications/Maximums

File Size: 2,000,000,000 bytes
Records per File: 1,000,000,000
Record Size: 4,000 bytes
Field Size
Character: 254 bytes
Date: 8 bytes
Logical: 1 byte
Numeric: 20 bytes including decimal point
Float: 20 bytes including decimal point
Memo: 64K (see note)
Fields per Record:512
Keys/Indexes per File:
.NDX: No Limit
.MDX 47 tags per .MDX files
Key Sizes
Character: 100 bytes
Numeric, Date: 8 bytes
Memo fields per File: Dependent on available memory
Open Files: Operating system dependent

dBaseIV:Driver Strings

There are switches or "driver strings" you can set to control the way your application creates, reads, and writes files with a specific driver. Driver strings are simply messages or parameters that are sent to the file driver at run-time to control its behavior. See *Common Driver Features--Driver Strings* for an overview of these runtime Database Driver switches and parameters.

Note:

Some driver strings have no effect after the file is open, so no SEND function syntax is listed for those strings. However, the SEND function syntax to return the value of the switch is listed for all driver strings.

The dBaseIV Driver supports the following Driver Strings:

BUFFERS

DRIVER('DBASE4', '/BUFFERS = n')

[Status" =] SEND(file, 'BUFFERS [= n]')

Sets the size of the buffer used to read and write to the file, where the buffer size is (n * 512 bytes). Use the /BUFFERS driver string to increase the buffer size if access is slow. Maximum buffer size 4,294,967,264. SEND returns the size of the buffer in bytes.

Tip

The default is three buffers of 1024 bytes each. Increasing the number of buffers will not increase performance when a file is shared by multiple users.

RECOVER

DRIVER('DBASE4', '/RECOVER')

[Status" =] SEND(file, 'RECOVER')

Equivalent to the Xbase RECALL command, which recovers records marked for deletion. When using the dBaseIV driver, the DELETE statement flags a record as "inactive." The driver does not remove the record until the PACK command is executed.

RECOVER is evaluated each time you open the file if you add the driver string to the data dictionary. When the driver recovers the records previously marked for deletion, you must manually rebuild keys and indexes with the BUILD statement.

IGNORESTATUS

DRIVER('DBASE4', '/IGNORESTATUS = on | off ')

[Status" =] SEND(file, 'IGNORESTATUS [on | off]')

When set *on*, the driver does *not* skip deleted records when accessing the file with GET, NEXT, and PREVIOUS in file order. It also enables a PUT on a deleted or held record. IGNORESTATUS requires opening the file in exclusive mode. SEND returns the IGNORESTATUS setting (ON or OFF) in the form of a STRING(3).

DELETED

[Status" =] SEND(file, 'DELETED')

For use only with the SEND command, when IGNORESTATUS is on. Returns the status of the current record. If deleted, the return string is "ON" and if not, "OFF."

ZEROY2K

DRIVER('DBASE4', '/ZEROY2K = on | off')

[Status" =] SEND(file, 'ZEROY2K [on | off]')

In the header of dBase4 files there is a field that stores the year that the file was last edited. Some applications store this as the number of years since 1900. Others store it as a 2 digit year. So for dates in the year 2000 some applications store 0 in this field and others 100. Clarion will read files with either. However it will write 100. Writing 100 may make the files unreadable by products that only support 0. To change this behavior you can with use a driver string of ZEROY2K, a SEND command or a setting in the WIN.INI file.

The driver will store 0 in the DBF file header when the WINI.INI setting is set to 1 or 'on' in a SEND command or driver string, otherwise a 100 will be stored in the DBF file header.

Note: The SEND command causes the setting to be set for all files that use that driver, not just for that file.

Example:

```
WIN.INI
;Sets all dBase4 files to store a 0 in the DBF file header
[CWDBASE4]
ZER0Y2K=1
!SEND command
SEND('Orders', ZER0Y2K='on' !sets Orders file to store 0 in the DBF file header
SEND('Orders', ZER0Y2K='off' !sets Orders file to store 100 in the DBF file header
!Driver String
Orders FILE, DRIVER('dbase4', '/ZER0Y2K=on'),PRE(ORD) !SETS Orders file to store 0
```

dBaseIV:Supported Commands and Attributes

File Attributes	Supported
CREATE	Y
DRIVER(<i>filetype</i> [, <i>driver string</i>])	Y
NAME	Y
ENCRYPT	N
OWNER(<i>password</i>)	N
RECLAIM	N ₁
PRE(<i>prefix</i>)	Y
BINDABLE	Y
THREAD	Y ₁₂
EXTERNAL(<i>member</i>)	Y
DLL(<i>[flag]</i>)	Y
OEM	Y
File Structures	Supported
INDEX	Y
KEY	Y
MEMO	Y
BLOB	N
RECORD	Y
Index, Key, Memo Attributes Supported	
BINARY	N
DUP	Y ₂
NOCASE	Y
OPT	N
PRIMARY	Y

NAME	Y
Ascending Components	Y
Descending Components	Y
Mixed Components	N
Field Attributes	Supported
DIM	N
OVER	Y
NAME	Y
File Procedures	Supported
BOF(<i>file</i>)	Y ₈
BUFFER(<i>file</i>)	N
BUILD(<i>file</i>)	Y
BUILD(<i>key</i>)	Y
BUILD(<i>index</i>)	Y
BUILD(<i>index, components</i>)	Y ₃
BUILD(<i>index, components, filter</i>)	N
BYTES(<i>file</i>)	N
CLOSE(<i>file</i>)	Y
COPY(<i>file, new file</i>)	Y ₄
CREATE(<i>file</i>)	Y
DUPLICATE(<i>file</i>)	Y
DUPLICATE(<i>key</i>)	Y
EMPTY(<i>file</i>)	Y
EOF(<i>file</i>)	Y ₈
FLUSH(<i>file</i>)	Y
LOCK(<i>file</i>)	N
NAME(<i>label</i>)	Y

OPEN(<i>file, access mode</i>)	Y5
PACK(<i>file</i>)	Y
POINTER(<i>file</i>)	Y9
POINTER(<i>key</i>)	Y9
POSITION(<i>file</i>)	Y10
POSITION(<i>key</i>)	Y10
RECORDS(<i>file</i>)	Y11
RECORDS(<i>key</i>)	Y11
REMOVE(<i>file</i>)	Y
RENAME(<i>file, new file</i>)	Y4
SEND(<i>file, message</i>)	Y
SHARE(<i>file, access mode</i>)	Y5
STATUS(<i>file</i>)	Y
STREAM(<i>file</i>)	Y
UNLOCK(<i>file</i>)	N
Record Access	Supported
ADD(<i>file</i>)	Y6
ADD(<i>file, length</i>)	N
APPEND(<i>file</i>)	Y6
APPEND(<i>file, length</i>)	N
DELETE(<i>file</i>)	Y1
GET(<i>file, key</i>)	Y
GET(<i>file, filepointer</i>)	Y
GET(<i>file, filepointer, length</i>)	N
GET(<i>key, keypointer</i>)	Y
HOLD(<i>file</i>)	Y7
NEXT(<i>file</i>)	Y

NOMEMO(<i>file</i>)	Y
PREVIOUS(<i>file</i>)	Y
PUT(<i>file</i>)	Y
PUT(<i>file</i> , <i>filepointer</i>)	Y
PUT(<i>file</i> , <i>filepointer</i> , <i>length</i>)	N
RELEASE(<i>file</i>)	Y
REGET(<i>file</i> , <i>string</i>)	Y
REGET(<i>key</i> , <i>string</i>)	Y
RESET(<i>file</i> , <i>string</i>)	Y
RESET(<i>key</i> , <i>string</i>)	Y
SET(<i>file</i>)	Y
SET(<i>file</i> , <i>key</i>)	Y
SET(<i>file</i> , <i>filepointer</i>)	Y
SET(<i>key</i>)	Y
SET(<i>key</i> , <i>key</i>)	Y
SET(<i>key</i> , <i>keypointer</i>)	Y
SET(<i>key</i> , <i>key</i> , <i>filepointer</i>)	Y
SKIP(<i>file</i> , <i>count</i>)	Y
WATCH(<i>file</i>)	Y
Transaction Processing	Supported
LOGOUT(<i>timeout</i> , <i>file</i> , ..., <i>file</i>)	N
COMMIT	N
ROLLBACK	N
Null Data Processing	Supported
NULL(<i>field</i>)	N
SETNULL(<i>field</i>)	N
SETNONNULL(<i>field</i>)	N

Notes

- 1 When the driver deletes a record from a dBase IV database, the record is not physically removed, instead the driver marks it inactive. Memo fields are not physically removed from the memo file, however they cannot be retrieved if they refer to an inactive record. Key values *are* removed from the index files. To remove records and memo fields permanently, execute a PACK(file).

Tip

To those programmers familiar with dBase IV, this driver processes deleted records consistent with the way dBase IV processes them after the SET DELETED ON command is issued. Records marked for deletion are ignored from processing by executable code statements, but remain in the data file.

- 2 In dBase IV it is legal to enter multiple records with duplicates of the unique key components. However, only the first of these records is indexed. So processing in key order only shows this first record. If you delete a record, then enter a new record with the same key value, the key file continues to point at the deleted record rather than the new record. In this situation, the dBase IV file driver driver changes the key file to point at the active record rather than the deleted record. This means that if you use a dBase IV program to delete a unique record, then insert a duplicate of this record, the new record is invisible when processing in key order until a pack is done. If you do the same process in a Clarion program, the new record is visible when processing in key order.

- 3 When building dynamic indexes, the *components* may take one of two forms:

```
BUILD(DynNdx, '+Pre:FLD1, -Pre:FLD2')
```

This form specifies the names of the fields on which to build the index. The field names must appear as specified in the fields' NAME() attribute if supplied, or must be the label name. A prefix may be used for compatibility with Clarion conventions but is ignored.

```
BUILD(DynNdx, 'T[Expression]')
```

This form specifies the type and Expression used to build an index--see *Miscellaneous--Key Definition* below.

- 4 These commands copy data and memo files using *newfile*, which may specify a new file name or directory. Key or index files are copied if the *newfile* is a subdirectory specification. To copy an index file to a new file, use a special form of the copy command:

```
COPY(file, '<index>|<newfile>')
```

This returns *File Not Found* if an invalid index is passed. The COPY command assumes a default extension of .NDX for both the source and the target file names if none is specified. If you require a file name without an extension, terminate the name with a period. Given the file structure:

```

Clar2 FILE,CREATE,DRIVER('dBase4'),PRE(CL2)
NumKey KEY(+CL2:Num),DUP
StrKey KEY(+CL2:Str1)
StrKey2 KEY(+CL2:Str2)
AMemo MEMO(100),NAME('mem')
Record RECORD
Num STRING(@n-9.2)
STR1 STRING(2)
STR2 STRING(2)
Mem STRING(10)
END
END

```

The following commands copy this file definition to A:

```

COPY(Clar2,'A:\CLAR2')
COPY(Clar2,'StrKey|A:\STRKEY')
COPY(Clar2,'StrKey2|A:\STRKEY2')
COPY(Clar2,'NumKey|A:\NUMKEY')

```

After these calls, the following files would exist on drive A: CLAR2.DBF, CLAR2.DBT, STRKEY.NDX, STRKEY2.NDX, and NUMKEY.NDX.

- 5 You do not need SHARE (or VSHARE) in any environment (for example, Novell) that supplies file locking as part of the operating system.
- 6 The ADD statement tests for duplicate keys before modifying the data file or its associated KEY files. Consequently it is slower than APPEND which performs no checks and does not update KEYS. When adding large amounts of data to a database use APPEND...BUILD in preference to ADD.
- 7 dBase IV performs record locking by locking the entire record within the data file. This prevents read access to other processes. Therefore we recommend minimizing the amount of time for which a record is held.
- 8 Although the driver supports these functions, we do not recommend their use. They must physically access the files and add overhead. Instead, test the value returned by ERRORCODE() after each sequential access. NEXT or PREVIOUS post Error 33 (Record Not Available) if an attempt is made to access a record beyond the end or beginning of the file.
- 9 There is no distinction between file pointers and key pointers; they both return the record number for any given record.

-
- 10 POSITION(file) returns a STRING(12). POSITION(key) returns a STRING containing the size of the key fields + 4 bytes.
 - 11 Under dBase IV, the RECORDS() function reports the same number of records for the data file and its keys and indexes. Usually there will be no difference in the number of records *unless* the INDEX is out of date. Because the DELETE statement does not physically remove records, *the number of records reported by the RECORDS() function includes inactive records*. Exercise care when using this function. The field names must appear as specified in the fields' NAME() attribute if supplied, or must be the label name. A prefix may be used for compatibility with the Clarion conventions but is ignored.
 - 12 THREADED files consume additional file handles for each thread that accesses the file.

dBaseIV:Other

International Sort Sequence

- † dBase IV sorts as if there were no diacritics in a field, thus A is sorted the same as Ä. If two words are identical except for diacritic characters, then the words are sorted as though the diacritic character was greater than the normal character. For example Äa < Ab < Äb whereas a CLADIGRAPH of ÄAE will sort as Ab < Äa < Äb. Solution- if the same file is used in Clarion and dBase IV, issue a BUILD statement rebuild the keys before updating the file (reading the file causes no problems).

Boolean Evaluation

- † dBase IV allows a logical field to accept one of 11 possible values (1,0,y,Y,n,N,t,T,f,F or a space character). The space character is neither true nor false. When using a logical field from a preexisting database in a logical expression, account for all these possibilities. Remember that when a STRING field is used as an expression, it is true if it contains any data and false if it is equal to zero or blank. Therefore, to evaluate a Logical field's truth, the expression should be true if the field contains any of the "true" characters (T,t,Y, or y). For example, if a Logical field were used to specify a product as taxable or nontaxable, the expression to evaluate its truth would be:

(If Condition):

```
Taxable='1' OR Taxable='T' OR Taxable='t' OR Taxable='Y' OR Taxable='y'
```

Large MEMOs

- † Clarion supports MEMO fields up to a maximum of 64K. If you have an existing file which includes a memo greater than 64K, you can use the file but not modify the large MEMOs.
- † You can determine when your application encounters a large MEMO by detecting when the memo pointer variable is non-blank, but the memo appears to be blank. Error 47 (Bad Record Declaration) is posted, and any modification to the MEMO field is ignored.

Long Field Names

- † dBase IV supports a maximum of 10 characters in a field name. If you require more, use an External Name with 10 characters or less.

Key Definition

- † dBase IV supports the use of expressions to define keys. Within the Dictionary Editor, you can place the expression in the external name field in the *Key Properties* dialog. The general format of the external name is :

`'FileName=T[Expression]'`

Where `FileName` represents the name of the index file (which can contain a path and file extension), and `T` represents the type of the index. Valid types are: C = character, D = date, and N = numeric. If the type is D or N then `Expression` can name only one field.

- † Multiple-index (.MDX) files require the NAME() attribute on a KEY or INDEX to specify the storage type of the key and any expression used to generate the key values. The general format of the NAME() attribute on a KEY or INDEX is:

`NAME('TagName|FileName[PageSize]=T[Expression],FOR[Expression]')`

The following documents the parameters for the NAME() attribute:

TagName	Specifies the name of an index tag within a multiple index file. If omitted the driver creates a dBase IV style .NDX file using the name specified in <code>FileName</code> .
FileName	Specifies the name of the index file, which may contain a path and extension.
PageSize	Specifies that when creating a .MDX file, (if a <code>TagName</code> is specified), a number in the range 2-32 specifying the number of 512-byte blocks in each index page. This value is only used when creating the file. If you specify multiple values with declarations for different tags in the same .MDX file, the largest value will be selected. The default value is 2.
T	Specifies the type of index. Legal types are C = character, D = date, N = numeric. If the type is D or N then <i>Expression</i> may name <i>only one</i> field.
Expression	Specifies an expression to generate the index. It may refer to multiple fields and invoke multiple xBase functions. The functions currently supported are listed below. Square brackets must enclose the expression.

Elements of the NAME() attribute may be omitted from the right. When specifying an `Expression`, you must also specify the type and name. If the `Expression` is omitted, the driver determines the `Expression` from the key fields when the file is created, or from the index file when opened.

If the type is omitted, the driver determines the index type from the first key component when the file is created, or from the index file when opened.

If the NAME() attribute is omitted altogether, the index file name is determined from the key label. The path defaults to the same location as the .DBF.

Tag names are limited to 9 characters in length. If the supplied name is too long it is automatically truncated.

Specify all field names in the NAME() attribute without a prefix.

- † dBase IV additionally supports the use of the Xbase FOR statement in expressions to define keys. The expressions supported in the FOR condition must be a simple condition of the form:

expression comparison_op expression

comparison_op may be <, <=, =<, <>, =, =>, >= or >.

The expression may refer to multiple fields in the record and contain xBase functions. Square brackets must enclose the expression. The currently supported functions appear below. If the driver encounters an unsupported Xbase function in a preexisting file, it posts error 76 'Invalid Index String' when the file is opened for keys and static indexes.

String expressions may use the '+' operator to concatenate multiple string arguments. Numeric expressions use the '+' or '-' operators with their conventional meanings. The maximum length of a dBase IV expression is 250 characters.

Supported xBase Key Definition Functions

ALLTRIM(string)	Removes leading and trailing spaces.
CTOD(string)	Converts a string key to a date. The <i>string</i> must be in the format mm/dd/yy; the result takes the form 'yyyymmdd'. The yyyy element of the date defaults to the twentieth century. An invalid date results in a key containing blanks.
DELETED()	Returns TRUE if the record is deleted.
DTOC(date)	Converts a date key to string format 'mm/dd/yy.'
DTOS(date)	Converts a date key to string format 'yyyymmdd.'
FIXED(float)	Converts a float key to a numeric.
FLOAT(numeric)	Converts a numeric key to a float.
IIF(bool, val1, val2)	Returns val1 if the first parameter is TRUE, otherwise returns val2.
LEFT(string, n)	Returns the leftmost <i>n</i> characters of the string key as a string of length <i>n</i> .
LOWER(string)	Converts a string key to lower case.
LTRIM(string)	Removes spaces from the left of a string.

RECNO()	Returns the current record number.
RIGHT(string, n)	Returns the rightmost <i>n</i> characters of the string key as a string of length <i>n</i> .
RTRIM(string)	Removes spaces from the right of a string.
STR(numeric [,length[, decimal places]])	Converts a numeric to a string. The length of the string and the number of decimal places are optional. The default string length is 10, and the number of decimal places is 0.
SUBSTR(string,offset,n)	Returns a substring of the <i>string</i> key starting at <i>offset</i> and of <i>n</i> characters in length.
TRIM(string)	Removes spaces from the right of a string (identical to RTRIM).
UPPER(string)	Converts a string key to upper case.
VAL(string)	Converts a string key to a numeric.

Copies the UNNAMED (the only table in the file) table from CUSTOMER.TPS (which has no password) to the CUSTOMER table in ORDERS.TPS which has the password acme96.

DOS Database Driver

DOS:Specifications

The DOS file driver reads and writes any binary, byte-addressable files. Neither fields nor records are delimited. When reading a record, the driver reads the number of bytes defined in the file's RECORD structure, unless a length parameter is specified in the GET statement.

The DOS driver supports the length parameter for the ADD, APPEND, GET, and PUT statements; this allows for variable length records in a DOS file.

The POINTER function returns the relative byte position within the file of the beginning of the last record accessed by an ADD, APPEND, GET, or NEXT statement.

This file driver performs forward sequential processing *only*. No key or transaction processing functions are supported, and the PREVIOUS statement is not supported.

Tip

Due to its limitations, the main function of this driver is as a disk editor for binary files.

Files:	C60DOSXL.LIB	Windows Static Link Library
	C60DOSX.LIB	Windows Export Library
	C60DOSX.DLL	Windows Dynamic Link Library (32-bit)

DOS:Data Types

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

DOS:File Specifications/Maximums

File Size:	4,294,967,295
Records per File :	4,294,967,295
Record Size:	64K
Field Size:	64K
Fields per Record:	64K
Keys/Indexes per File:	n/a
Key Size:	n/a
Memo fields per File:	n/a
Memo Field Size :	n/a
Open Data Files :	Operating system dependent

DOS:Driver Strings

There are switches or "driver strings" you can set to control the way your application creates, reads, and writes files with a specific driver. Driver strings are simply messages or parameters that are sent to the file driver at run-time to control its behavior. See *Common Driver Features--Driver Strings* for an overview of these runtime Database Driver switches and parameters.

Note:

Some driver strings have no effect after the file is open, so no SEND function syntax is listed for those strings. However, the SEND function syntax to return the value of the switch is listed for all driver strings.

The DOS Driver supports the following Driver Strings:

FILEBUFFERS

DRIVER('DOS', '/FILEBUFFERS = n')

[Buffers" =] SEND(file, 'FILEBUFFERS [= n]')

Sets the size of the buffer used to read and write to the file, where the buffer size is (n * 512 bytes). Use the /FILEBUFFERS driver string to increase the buffer size if access is slow. Maximum buffer size is 4,294,967,264. SEND returns the size of the buffer in bytes.

Tip

The default buffer size for files opened denying write access to other users is the larger of 1024 or (2 * record size), and the larger of 512 or record size for all other open modes.

QUICKSCAN

DRIVER('DOS', '/QUICKSCAN = on | off')

[QScan" =] SEND(file, 'QUICKSCAN [= on | off]')

Specifies buffered access behavior. The ASCII driver reads a buffer at a time (not a record), allowing faster access. In a multi-user environment these buffers are not 100% trustworthy for subsequent access, because another user may change the file between accesses. As a safeguard, the driver rereads the buffers before each record access. To *disable* the reread, set QUICKSCAN to ON. The default is ON for files opened denying write access to other users, and OFF for all other open modes. SEND returns the Quicksan setting (ON or OFF) in the form of a STRING(3).

DOS:Supported Commands and Attributes

File Attributes	Supported
CREATE	Y
DRIVER(<i>filetype</i> [, <i>driver string</i>])	Y
NAME	Y
ENCRYPT	N
OWNER(<i>password</i>)	N
RECLAIM	N
PRE(<i>prefix</i>)	Y
BINDABLE	Y
THREAD	Y4
EXTERNAL(<i>member</i>)	Y
DLL(<i>flag</i>)	Y
OEM	Y
File Structures	Supported
INDEX	N
KEY	N

MEMO	N
BLOB	N
RECORD	Y
Index, Key, Memo Attributes	Supported
BINARY	N
DUP	N
NOCASE	N
OPT	N
PRIMARY	N
NAME	N
Ascending Components	N
Descending Components	N
Mixed Components	N
Field Attributes	Supported
DIM	Y
OVER	Y
NAME	Y
File Procedures	Supported
BOF(<i>file</i>)	N
BUFFER(<i>file</i>)	N
BUILD(<i>file</i>)	N
BUILD(<i>key</i>)	N
BUILD(<i>index</i>)	N
BUILD(<i>index, components</i>)	N
BUILD(<i>index, components, filter</i>)	N
BYTES(<i>file</i>)	Y
CLOSE(<i>file</i>)	Y

COPY(<i>file, new file</i>)	Y
CREATE(<i>file</i>)	Y
DUPLICATE(<i>file</i>)	N
DUPLICATE(<i>key</i>)	N
EMPTY(<i>file</i>)	Y
EOF(<i>file</i>)	Y
FLUSH(<i>file</i>)	N
LOCK(<i>file</i>)	Y
NAME(<i>label</i>)	Y
OPEN(<i>file, access mode</i>)	Y
PACK(<i>file</i>)	N
POINTER(<i>file</i>)	Y ₂
POINTER(<i>key</i>)	N
POSITION(<i>file</i>)	Y ₃
POSITION(<i>key</i>)	N
RECORDS(<i>file</i>)	Y
RECORDS(<i>key</i>)	N
REMOVE(<i>file</i>)	Y
RENAME(<i>file, new file</i>)	Y
SEND(<i>file, message</i>)	Y
SHARE(<i>file, access mode</i>)	Y
STATUS(<i>file</i>)	Y
STREAM(<i>file</i>)	N
UNLOCK(<i>file</i>)	Y
Record Access	Supported
ADD(<i>file</i>)	Y
ADD(<i>file, length</i>)	Y

APPEND(<i>file</i>)	Y
APPEND(<i>file, length</i>)	Y
DELETE(<i>file</i>)	N
GET(<i>file, key</i>)	N
GET(<i>file, filepointer</i>)	Y
GET(<i>file, filepointer, length</i>)	Y
GET(<i>key, keypointer</i>)	N
HOLD(<i>file</i>)	N
NEXT(<i>file</i>)	Y
NOMEMO(<i>file</i>)	N
PREVIOUS(<i>file</i>)	Y
PUT(<i>file</i>)	Y
PUT(<i>file, filepointer</i>)	Y ₁
PUT(<i>file, filepointer, length</i>)	Y ₁
RELEASE(<i>file</i>)	N
REGET(<i>file, string</i>)	Y
REGET(<i>key, string</i>)	N
RESET(<i>file, string</i>)	Y
RESET(<i>key, string</i>)	N
SET(<i>file</i>)	Y
SET(<i>file, key</i>)	N
SET(<i>file, filepointer</i>)	Y
SET(<i>key</i>)	N
SET(<i>key, key</i>)	N
SET(<i>key, keypointer</i>)	N
SET(<i>key, key, filepointer</i>)	N
SKIP(<i>file, count</i>)	N

WATCH(<i>file</i>)	N
Transaction Processing	Supported
LOGOUT(<i>timeout, file, ..., file</i>)	N
COMMIT	N
ROLLBACK	N
Null Data Processing	Supported
NULL(<i>field</i>)	N
SETNULL(<i>field</i>)	N
SETNONNULL(<i>field</i>)	N

Notes

- 1 When using PUT() with this driver you should take care to PUT back the same number of characters that were read. If you PUT back more characters than were read, then the "extra" characters will overwrite the first part of the subsequent record. If you PUT back fewer characters than were read, then only the first part of the retrieved record is overwritten, while the last part of the retrieved record remains as it was prior to the PUT().
- 2 POINTER() returns the relative byte position within the file.
- 3 POSITION(file) returns a STRING(4).
- 4 THREADED files consume additional file handles for each thread that accesses the file.

FoxPro / FoxBase Database Driver

FoxPro:Specifications

The FoxPro file driver is compatible with FoxPro and FoxBase. The default data file extension is *.DBF.

The default index file extension is *.IDX. The default Memo file extension is .FBT. FoxPro also supports multiple index files, whose default extension is *.CDX. The miscellaneous section describes the procedures for using the .CDX files.

Files:	C60FOXXL.LIB	Windows Static Link Library
	C60FOXX.LIB	Windows Export Library
	C60FOXX.DLL	Windows Dynamic Link Library

Tip

The FoxPro index file format is the backbone of its vaunted "Rushmore" technology. The old saying "There's no free lunch," however, applies. Adding and appending records to a large database is a slower process than in other xBase formats, due to the time required to update the index file.

FoxPro:Data Types

The xBase file format stores all data as ASCII strings. You may either specify STRING types with declared pictures for each field, or specify native Clarion types, which the driver converts automatically.

<u>FoxPro data type</u>	<u>Clarion data type</u>	<u>STRING w/ picture</u>
Date	DATE	STRING(@D12)
*Numeric	REAL	STRING(@N-_p.d)
*Logical	BYTE	STRING(1)
Character	STRING	STRING
*Memo	MEMO	MEMO

If your application reads and writes to existing files, a pictured STRING will suffice. However, if your application *creates* a FoxPro or FoxBase file, you may require additional information for these FoxPro and FoxBase types:

- † To create a numeric field in the Data Dictionary, choose the REAL data type. In the External Name field on the Attributes tab, specify '*NumericFieldName*=N(*Precision*,*DecimalPlaces*)' where *NumericFieldName* is the name of the field, *Precision* is the precision of the field and *DecimalPlaces* is the number of decimal places. With a REAL data type, you cannot access the Character or Places fields in the Field definition, you must specify those attributes with an expression in the External Name Field on the Attributes tab.

For example, if you want to create a field called Number with nine significant digits and two decimal places, enter 'Number=N(9,2) in the External Name field on the Attributes tab of the Field properties in the Data Dictionary.

If you're hand coding a native Clarion data type, add the NAME attribute using the same syntax.

If you're hand coding a STRING with picture, STRING(@N-_9.2), NAME('Number'), where *Number* is the field name.

- † To create a logical field, using the data dictionary, choose the BYTE data type. There are no special steps; however, see the miscellaneous section for tips on reading the data from the field.

If you're hand coding a STRING with picture, add the NAME attribute: STRING(1), NAME('LogFld = L').

- † To create a date field, using the data dictionary, choose the DATE data type, rather than LONG, which you usually use for the TopSpeed or Clarion file formats.

- † MEMO field declarations require the a pointer field in the file's record structure. Declare the pointer field as a STRING(10) or a LONG. This field will be stored in the .DBF file containing the offset of the memo in the .DBT file. The MEMO declaration must have a NAME() attribute naming the pointer field. An example file declaration follows:

```
File FILE, DRIVER('FoxPro')
Memo1 MEMO(200),NAME('Notes')
Memo2 MEMO(200),NAME('Text')
Rec RECORD
Mem1Ptr LONG,NAME('Notes')
Mem2Ptr STRING(10),NAME('Text')
END
END
```

FoxPro:File Specifications/Maximums

File Size: 2,000,000,000 bytes
Records per File: 1,000,000,000 bytes
Record Size: 4,000 bytes
Field Size
Character: 254 bytes
Date: 8 bytes
Logical: 1 byte
Numeric: 20 bytes including decimal point
Float: 20 bytes including decimal point
Memo: 65,520 bytes (see note)
Fields per Record: 512
Keys/Indexes per File: No Limit
Key Sizes
Character: 100 bytes (.IDX)
254 bytes (.CDX)
Numeric, Date: 8 bytes
Memo fields per File: Dependent on available memory
Open Files: Operating system dependent

FoxPro:Driver Strings

There are switches or "driver strings" you can set to control the way your application creates, reads, and writes files with a specific driver. Driver strings are simply messages or parameters that are sent to the file driver at run-time to control its behavior. See *Common Driver Features--Driver Strings* for an overview of these runtime Database Driver switches and parameters.

Note:

Some driver strings have no effect after the file is open, so no SEND function syntax is listed for those strings. However, the SEND function syntax to return the value of the switch is listed for all driver strings.

The FoxPro Driver supports the following Driver Strings:

BUFFERS

DRIVER('FOXPRO', '/BUFFERS = n')

[Status" =] SEND(file, 'BUFFERS [= n]')

Sets the size of the buffer used to read and write to the file, where the buffer size is (n * 512 bytes). Use the /BUFFERS driver string to increase the buffer size if access is slow. Maximum buffer size is 4,294,967,264. SEND returns the size of the buffer in bytes.

Tip

The default is three buffers of 1024 bytes each. Increasing the number of buffers will not increase performance when a file is shared by multiple users.

RECOVER

DRIVER('FOXPRO', '/RECOVER')

[Status" =] SEND(file, 'RECOVER')

Equivalent to the Xbase RECALL command, which recovers records marked for deletion. When using the FoxPro driver, the DELETE statement flags a record as "inactive." The driver does not remove the record until the PACK command is executed.

RECOVER is evaluated each time you open the file if you add the driver string to the data dictionary. When the driver recovers the records previously marked for deletion, you must manually rebuild keys and indexes with the BUILD statement.

IGNORESTATUS

```
DRIVER('FOXPRO', '/IGNORESTATUS = on | off ' )
[ Status" = ] SEND(file, 'IGNORESTATUS [ on | off ] ' )
```

When set *on*, the driver does *not* skip deleted records when accessing the file with GET, NEXT, and PREVIOUS in file order. It also enables a PUT on a deleted or held record. IGNORESTATUS requires opening the file in exclusive mode. SEND returns the IGNORESTATUS setting (ON or OFF) in the form of a STRING(3).

DELETED

```
[ Status" = ] SEND(file, 'DELETED' )
```

For use only with the SEND command, when IGNORESTATUS is on. Returns the status of the current record. If deleted, the return string is "ON" and if not, "OFF."

ZEROY2K

```
DRIVER('FOXPRO', '/ZEROY2K = on | off' )
[ Status" = ] SEND(file, 'ZEROY2K [ on | off ] ' )
```

In the header of FoxPro files there is a field that stores the year that the file was last edited. Some applications store this as the number of years since 1900. Others store it as a 2 digit year. So for dates in the year 2000 some applications store 0 in this field and others 100. Clarion will read files with either. However it will write 100. Writing 100 may make the files unreadable by products that only support 0. To change this behavior you can with use a driver string of ZEROY2K, a SEND command or a setting in the WIN.INI file.

The driver will store 0 in the DBF file header when the WINI.INI setting is set to 1 or 'on' in a SEND command or driver string, otherwise a 100 will be stored in the DBF file header.

Note: The SEND command causes the setting to be set for all files that use that driver, not just for that file.

Example:

```
WIN.INI
;Sets all FoxPro files to store a 0 in the DBF file header
[CWFOXPRO]
ZEROY2K=1
!SEND command
SEND('Orders', ZEROY2K='on' !sets Orders file to store 0 in the DBF file header
SEND('Orders', ZEROY2K='off' !sets Orders file to store 100 in the DBF file header
!Driver String
Orders FILE, DRIVER('FOXPRO', '/ZEROY2K=on'),PRE(ORD) !SETS Orders file to store 0
```

FoxPro:Supported Commands and Attributes

File Attributes	Supported
CREATE	Y
DRIVER(<i>filetype</i> [, <i>driver string</i>])	Y
NAME	Y
ENCRYPT	N
OWNER(<i>password</i>)	N
RECLAIM	N ₁
PRE(<i>prefix</i>)	Y
BINDABLE	Y
THREAD	Y ₁₃
EXTERNAL(<i>member</i>)	Y
DLL(<i>[flag]</i>)	Y
OEM	N ₂
File Structures	Supported
INDEX	Y
KEY	Y
MEMO	Y
BLOB	N
RECORD	Y
Index, Key, Memo Attributes	Supported
BINARY	N ₁₄
DUP	Y ₃
NOCASE	Y
OPT	N
PRIMARY	Y
NAME	Y

Ascending Components	Y
Descending Components	Y
Mixed Components	N
Field Attributes	Supported
DIM	N
OVER	Y
NAME	Y
File Procedures	Supported
BOF(<i>file</i>)	Y ₉
BUFFER(<i>file</i>)	N
BUILD(<i>file</i>)	Y
BUILD(<i>key</i>)	Y
BUILD(<i>index</i>)	Y
BUILD(<i>index, components</i>)	Y ₄
BUILD(<i>index, components, filter</i>)	N
BYTES(<i>file</i>)	N
CLOSE(<i>file</i>)	Y
COPY(<i>file, new file</i>)	Y ₅
CREATE(<i>file</i>)	Y
DUPLICATE(<i>file</i>)	Y
DUPLICATE(<i>key</i>)	Y
EMPTY(<i>file</i>)	Y
EOF(<i>file</i>)	Y ₉
FLUSH(<i>file</i>)	Y
LOCK(<i>file</i>)	N
NAME(<i>label</i>)	Y
OPEN(<i>file, access mode</i>)	Y ₆

PACK(<i>file</i>)	Y
POINTER(<i>file</i>)	Y ₁₀
POINTER(<i>key</i>)	Y ₁₀
POSITION(<i>file</i>)	Y ₁₁
POSITION(<i>key</i>)	Y ₁₁
RECORDS(<i>file</i>)	Y ₁₂
RECORDS(<i>key</i>)	Y ₁₂
REMOVE(<i>file</i>)	Y
RENAME(<i>file, new file</i>)	Y ₅
SEND(<i>file, message</i>)	Y
SHARE(<i>file, access mode</i>)	Y ₆
STATUS(<i>file</i>)	Y
STREAM(<i>file</i>)	Y
UNLOCK(<i>file</i>)	Y
Record Access	Supported
ADD(<i>file</i>)	Y ₇
ADD(<i>file, length</i>)	N
APPEND(<i>file</i>)	Y ₇
APPEND(<i>file, length</i>)	N
DELETE(<i>file</i>)	Y ₁
GET(<i>file, key</i>)	Y
GET(<i>file, filepointer</i>)	Y
GET(<i>file, filepointer, length</i>)	N
GET(<i>key, keypointer</i>)	Y
HOLD(<i>file</i>)	Y ₈
NEXT(<i>file</i>)	Y
NOMEMO(<i>file</i>)	Y

PREVIOUS(<i>file</i>)	Y
PUT(<i>file</i>)	Y
PUT(<i>file</i> , <i>filepointer</i>)	Y
PUT(<i>file</i> , <i>filepointer</i> , <i>length</i>)	N
RELEASE(<i>file</i>)	Y
REGET(<i>file</i> , <i>string</i>)	Y
REGET(<i>key</i> , <i>string</i>)	Y
RESET(<i>file</i> , <i>string</i>)	Y
RESET(<i>key</i> , <i>string</i>)	Y
SET(<i>file</i>)	Y
SET(<i>file</i> , <i>key</i>)	Y
SET(<i>file</i> , <i>filepointer</i>)	Y
SET(<i>key</i>)	Y
SET(<i>key</i> , <i>key</i>)	Y
SET(<i>key</i> , <i>keypointer</i>)	Y
SET(<i>key</i> , <i>key</i> , <i>filepointer</i>)	Y
SKIP(<i>file</i> , <i>count</i>)	Y
WATCH(<i>file</i>)	Y
Transaction Processing	Supported
LOGOUT(<i>timeout</i> , <i>file</i> , ..., <i>file</i>)	N
COMMIT	N
ROLLBACK	N
Null Data Processing	Supported
NULL(<i>field</i>)	N
SETNULL(<i>field</i>)	N
SETNONNULL(<i>field</i>)	N

Notes

- 1 When the driver deletes a record from a FoxPro database, the record is not physically removed, instead the driver marks it inactive. Memo fields are not physically removed from the memo file, however they cannot be retrieved if they refer to an inactive record. Key values *are* removed from the index files. To remove records and memo fields permanently, execute a PACK(file).

Tip

To those programmers familiar with FoxPro, this driver processes deleted records consistent with the way FoxPro processes them after the SET DELETED ON command is issued. Records marked for deletion are ignored from processing by executable code statements, but remain in the data file.

- 2 If you need to access FoxPro data with alternate characters stored using a non-English version of FoxPro, then you should use ODBC. However, if you do not have any string based kesys, you can use the FoxPro driver and call the ConvertOEMToANSI and ConvertANSIToOEM after retrieving and before updating a record.
- 3 In FoxPro it is legal to enter multiple records with duplicates of the unique key components. However, only the first of these records is indexed. So processing in key order only shows this first record. If you delete a record, then enter a new record with the same key value, the key file continues to point at the deleted record rather than the new record. In this situation, the FoxPro file driver changes the key file to point at the active record rather than the deleted record. This means that if you use a FoxPro program to delete a unique record, then insert a duplicate of this record, the new record is invisible when processing in key order until a pack is done. If you do the same process in a Clarion program, the new record is visible when processing in key order.
- 4 When building dynamic indexes, the *components* may take one of two forms:

```
BUILD(DynNdx, '+Pre:FLD1, -Pre:FLD2')
```

This form specifies the names of the fields on which to build the index. The field names must appear as specified in the fields' NAME() attribute if supplied, or must be the label name. A prefix may be used for compatibility with Clarion conventions but is ignored.

```
BUILD(DynNdx, 'T[Expression]')
```

This form specifies the type and Expression used to build an index--see *Miscellaneous--Key Definition* for more information.

- 5 These commands copy data and memo files using *newfile*, which may specify a new file name or directory. Key or index files are copied if the *newfile* is a subdirectory specification. To copy an index file to a new file, use a special form of the copy command:

```
COPY(file, '<index>|<newfile>')
```

This returns *File Not Found* if an invalid index is passed. The COPY command assumes a default extension of .IDX for both the source and the target file names if none is specified. If you require a file name without an extension, terminate the name with a period. Given the file structure:

```

Clar2  FILE,CREATE,DRIVER('FoxPro'),PRE(CL2)
NumKey KEY(+CL2:Num),DUP
StrKey KEY(+CL2:Str1)
StrKey2 KEY(+CL2:Str2)
AMemo  MEMO(100),NAME('mem')
Record RECORD
Num    STRING(@n-_9.2)
STR1   STRING(2)
STR2   STRING(2)
Mem    STRING(10)
      END
      END

```

The following commands copy this file definition to A:

```

COPY(Clar2,'A:\CLAR2')
COPY(Clar2,'StrKey|A:\STRKEY')
COPY(Clar2,'StrKey2|A:\STRKEY2')
COPY(Clar2,'NumKey|A:\NUMKEY')

```

After these calls, the following files would exist on drive A: CLAR2.DBF, CLAR2.FPT, STRKEY.IDX, STRKEY2.IDX, and NUMKEY.IDX.

- 6 You do not need SHARE (or VSHARE) in any environment (for example, Novell) that supplies file locking as part of the operating system.
- 7 The ADD statement tests for duplicate keys before modifying the data file or its associated KEY files. Consequently it is slower than APPEND which performs no checks and does not update KEYs. When adding large amounts of data to a database use APPEND...BUILD in preference to ADD.
- 8 FoxPro performs record locking by locking the entire record within the data file. This prevents read access to other processes. Therefore we recommend minimizing the amount of time for which a record is held.
- 9 Although the driver supports these functions, we do not recommend their use. They must physically access the files and they are slow. Instead, test the value returned by ERRORCODE() after each sequential access. NEXT or PREVIOUS post Error 33 (Record Not Available) if an attempt is made to access a record beyond the end or beginning of the file.
- 10 There is no distinction between file pointers and key pointers; they both return the record number for the given record.

- 11 POSITION(file) returns a STRING(12). POSITION(key) returns a STRING the size of the key fields + 4 bytes.
- 12 Under FoxPro, the RECORDS() function reports the same number of records for the data file and its keys and indexes. Usually there will be no difference in the number of records *unless* the INDEX is out of date. Because the DELETE statement does not physically remove records, *the number of records reported by the RECORDS() function includes inactive records*. Exercise care when using this function. The field names must appear as specified in the fields' NAME() attribute if supplied, or must be the label name. A prefix may be used for compatibility with the Clarion conventions but is ignored.
- 13 THREADED files consume additional file handles for each thread that accesses the file.
- 14 OEM conversion is not applied to BINARY MEMOs. The driver assumes BINARY MEMOs are zero padded; otherwise, space padded.

FoxPro:Other

Boolean Evaluation

- † FoxPro and FoxBase allow a logical field to accept one of 11 possible values (0,1,y,Y,n,N,t,T,f,F or a space character). The space character is neither true nor false. When using a logical field from a preexisting database in a logical expression, account for all these possibilities. Remember that when a STRING field is used as an expression, it is true if it contains any data and false if it is equal to zero or blank. Therefore, to evaluate a Logical field's truth, the expression should be true if the field contains any of the "true" characters (1,T,t,Y, or y). For example, if a Logical field were used to specify a product as taxable or nontaxable, the expression to evaluate its truth would be:

(If Condition):

```
Taxable='1' OR Taxable='T' OR Taxable='t' OR Taxable='Y' OR Taxable='y'
```

Large MEMOs

- † Clarion supports MEMO fields up to a maximum of 64K. If you have an existing file which includes a memo greater than 64K, you can use the file but not modify the large MEMOs.
- † You can determine when your application encounters a large MEMO by detecting when the memo pointer variable is non-blank, but the memo appears to be blank. Error 47 (Bad Record Declaration) is posted, and any modification to the MEMO field is ignored.

Long Field Names

- † FoxPro and FoxBase support a maximum of 10 characters in a field name. If you require more, use an External Name with 10 characters or less.

Key Definition

- † FoxPro and FoxBase support the use of expressions to define keys. Within the Dictionary Editor, you can place the expression in the external name field in the *Key Properties* dialog. The general format of the external name is :

```
'FileName=T[Expression]'
```

Where **FileName** represents the name of the index file (which can contain a path and file extension), and **T** represents the type of the index. Valid types are: C = character, D = date, and N = numeric. If the type is D or N then **Expression** can name only one field.

- † Multiple-index (.CDX) files require the NAME() attribute on a KEY or INDEX to specify the storage type of the key and any expression used to generate the key values. The general format of the NAME() attribute on a KEY or INDEX is:

NAME('TagName | FileName=T[Expression] ,COMPRESSED')

The following are the parameters for the NAME() attribute:

TagName	Names an index tag within a multiple index file. If the TagName is omitted the driver creates an .IDX file with the name specified in FileName.
FileName	Names the index file, and optionally contains a path and extension.
T	Specifies the type of the index; legal types are C = character, D = date, N = numeric. If the type is D or N then <i>Expression</i> may name only one field.
Expression	Specifies the expression used to generate the index. The expression may refer to multiple fields, and invoke multiple of xBase functions. The functions currently supported are listed below. Square brackets must enclose the expression.
COMPRESSED	When specified, the FoxPro Driver creates a FoxPro 2 compatible compressed .IDX file.

Elements of the NAME() attribute may be omitted from the right. When specifying an Expression, the type and name must also be specified. If the Expression is omitted, the driver determines the Expression from the key fields when the file is created, or from the index file when opened.

If the type is omitted, the driver determines the index type from the first key component when the file is created, or from the index file when opened.

If the NAME() attribute is omitted altogether, the index file name is determined from the key label. The path defaults to the same location as the .DBF.

Tag names are limited to 10 characters in length; if the supplied name is too long it is automatically truncated.

All field names in the NAME() attribute must be specified without a prefix.

- † FoxPro additionally supports the use of the Xbase FOR statement in expressions to define keys. The expressions supported in the FOR condition must be a simple condition of the form:

`expression comparison_op expression`

`comparison_op` may be one of the following: <, <=, =<, <>, =, =>, >= or >.

The expression may refer to multiple fields in the record, and contain xBase functions. Square brackets must enclose the expression. The currently supported functions appear below. If the driver encounters an unsupported Xbase function in a preexisting file, it posts error 76 'Invalid Index String' when the file is opened for keys and static indexes.

String expressions may use the '+' operator to concatenate multiple string arguments. Numeric expressions use the '+' or '-' operators with their conventional meanings. The maximum length of a FoxPro or FoxBase expression is 250 characters.

Supported xBase Key Definition Functions

<code>ALLTRIM(string)</code>	Removes leading and trailing spaces.
<code>CTOD(string)</code>	Converts a string key to a date. The <i>string</i> must be in the format mm/dd/yy; the result takes the form 'yyyymmdd'. The yyyy element of the date defaults to the twentieth century. An invalid date results in a key containing blanks.
<code>DELETED()</code>	Returns TRUE if the record is deleted.
<code>DTOC(date)</code>	Converts a date key to string format 'mm/dd/yy.'
<code>DTOS(date)</code>	Converts a date key to string format 'yyyymmdd.'
<code>FIXED(float)</code>	Converts a float key to a numeric.
<code>FLOAT(numeric)</code>	Converts a numeric key to a float.
<code>IIF(bool, val1, val2)</code>	Returns val1 if the first parameter is TRUE, otherwise returns val2.
<code>LEFT(string, n)</code>	Returns the leftmost <i>n</i> characters of the string key as a string of length <i>n</i> .
<code>LOWER(string)</code>	Converts a string key to lower case.
<code>LTRIM(string)</code>	Removes spaces from the left of a string.
<code>RECNO()</code>	Returns the current record number.
<code>RIGHT(string, n)</code>	Returns the rightmost <i>n</i> characters of the string key as a string of length <i>n</i> .
<code>RTRIM(string)</code>	Removes spaces from the right of a string.

<code>STR(numeric [,length[, decimal places]])</code>	Converts a numeric to a string. The length of the string and the number of decimal places are optional. The default string length is 10, and the number of decimal places is 0.
<code>SUBSTR(string,offset,n)</code>	Returns a substring of the <i>string</i> key starting at <i>offset</i> and of <i>n</i> characters in length.
<code>TRIM(string)</code>	Removes spaces from the right of a string (identical to RTRIM).
<code>UPPER(string)</code>	Converts a string key to upper case.
<code>VAL(string)</code>	Converts a string key to a numeric.

TopSpeed Database Driver

TopSpeed:Overview

The TopSpeed Database file system is a high-performance, high-security, proprietary file driver for Clarion development tools. It is *not* file compatible with the Clarion file driver's data.

Data tables, keys, indexes and memos can all be stored together in a single DOS file. The default file extension is *.TPS. A separate "Transaction Control File" uses the *.TCF extension by default.

The TopSpeed driver can optionally store multiple tables in a single file. This lets you open as many data tables, keys, and indexes as necessary using a *single DOS file handle*. This feature may be especially useful when there are a large number of small tables, or when a group of related files are normally accessed together. All keys, indexes, and memos are stored internally.

Tip

When multiple tables share a single DOS handle, the first OPEN mode applies to all the tables within the file.

In addition, the TopSpeed file system supports the **BLOB** data type (Binary Large Object), a field which is completely variable-length and may be greater than 64K in size. A BLOB must be declared before the RECORD structure. Memory for a BLOB is dynamically allocated and de-allocated as necessary. For more information, see *BLOB* in the *Language Reference*.

Files:	C60TPSXL.LIB	Windows Static Link Library
	C60TPSX.LIB	Windows Export Library
	C60TPSX.DLL	Windows Dynamic Link Library

Tip

This driver offers speed, security, and takes up fewer resources on the end user's system.

TopSpeed:Data Types

BYTE	DECIMAL
SHORT	STRING
USHORT	CSTRING
LONG	PSTRING
ULONG	MEMO
SREAL	GROUP
REAL	BLOB
DATE	TIME

TopSpeed:File Maximums/Specifications

```

File Size :                2 GB
Records per File :        Unsigned Long (4,294,967,295)
Record Size :             15,000 bytes
Field Size :              15,000 bytes
Fields per Record :      15,000
Keys/Indexes per File:   240
Key Size :                15,000 bytes
Memo fields per File:    255
Memo Field Size :        64,000 bytes
BLOB fields per File:    255
BLOB Size :               Hardware dependent (Max size 640 MB)
Open Data Files :        Operating system dependent
Table Name :              1,000 bytes
Tables per DOS File :    Limited only by the maximum DOS
                        file size--approximately 2^32
                        bytes (4,294,967,296).
Concurrent Users per File: 1024

```

TopSpeed:Driver Strings

There are switches or "driver strings" you can set to control the way your application creates, reads, and writes files with a specific driver. Driver strings are simply messages or parameters that are sent to the file driver at run-time to control its behavior. See *Common Driver Features--Driver Strings* for an overview of these runtime Database Driver switches and parameters.

Note:

Some driver strings have no effect after the file is open, so no SEND function syntax is listed for those strings. However, the SEND function syntax to return the value of the switch is listed for all driver strings.

The TopSpeed Driver supports the following Driver Strings:

DECIMALCheck

```
[ DECCheck" = ] SEND(file, 'DECIMALCheck [ = ON|OFF ]')
```

DECIMALCheck=OFF ensures compatibility of TopSpeed files originally created in certain early versions of Clarion with the current version by disabling error reporting for the number of decimal places in DECIMAL fields during file header comparisons.

This switch should be used as a driver string, or, in a SEND command before the file is opened.

FLAGS

```
[ Flags" = ] SEND(file, 'FLAGS [ = bitmap ]' )
```

Sets and returns the configuration flags for the *file*. Use the following EQUATES declared in EQUATES.CLW to control the behavior of the target TopSpeed file:

```
!TopSpeed File Flags
```

```
TPSREADONLY EQUATE(1)
```

For example, the following code makes the file read-only for ODBC access while preserving any other flags:

```
TpsFlags = SEND(MyFile, 'FLAGS')
SEND(MyFile, 'FLAGS = '&BOR(TpsFlags,TPSREADONLY)
```

FULLBUILD

```
DRIVER('TOPSPEED', '/FULLBUILD = on | off' )
```

```
[ State" = ] SEND(file, 'FULLBUILD [ = on | off ]' )
```

```
[ State" = ] file{PROP:FULLBUILD} [ = on | off ]' )
```

The TopSpeed driver has an optimized appending mechanism where you can add large numbers of records to an existing table with the APPEND statement. Issuing a subsequent BUILD updates only the appended key information, making incremental batch updates very fast. This is the default behavior. Use the FULLBUILD driver string to modify this default behavior.

FULLBUILD=ON tells the next BUILD statement to fully rebuild the keys. FULLBUILD=OFF restores the BUILD to its optimized state. Both versions of the SEND command return the current build state as a string 'ON' or 'OFF'. Issue SEND(file,'FULLBUILD') to return the current build state without changing it.

PNM=

```
TName" = SEND(file, 'PNM=[starting point]' )
```

Returns the next table name in the *file's* TopSpeed super file, after the specified *starting point*. If there are no table names after the specified *starting point*, SEND returns an empty string. If *starting point* is omitted or contains an empty string, SEND returns the first table name in the file. PNM= is only valid with the SEND command. There are no spaces surrounding the equal sign (=). The target *file* is the label of any of the tables within the TopSpeed super file.

For example, given a TopSpeed file containing the Supp table, the following code displays an alphabetical listing of all the tables in the file:

```

CODE
name = ''
LOOP
  name = SEND(Supp, 'PNM=' & name)
  If name
    MESSAGE(name)
  ELSE
    BREAK
  END
END
END

```

TCF

```
DRIVER('TOPSPEED', '/TCF = filename' )
```

```
[TCFPath =] SEND(file, 'TCF [= filename]' )
```

Specifies a transaction control file other than the default \TOPSPEED.TCF. The file identifies all transactions in progress until the program terminates or a SEND(file, 'TCF = filename') executes. In other words, the TCF setting affects all TopSpeed files accessed by the program. This returns the name of the transaction control file. For example, TCFPath = SEND(file, 'TCF').

Note:

We recommend using one transaction control file for a system. Using multiple files with different access rights can result in partially committed transactions-- some of the files within a transaction might be updated and others left unchanged.

See Transaction Control Files for more information on implementing this technique.

TopSpeed:Supported Commands and Attributes

File Attributes	Supported
CREATE	Y
DRIVER(<i>filetype</i> [, <i>driver string</i>])	Y
NAME	Y
ENCRYPT	Y
OWNER(<i>password</i>)	Y ₁
RECLAIM	N ₂
PRE(<i>prefix</i>)	Y
BINDABLE	Y

THREAD	Y ₁₂
EXTERNAL(<i>member</i>)	Y
DLL(<i>flag</i>)	Y
OEM	Y
File Structures	Supported
INDEX	Y
KEY	Y
MEMO	Y ₃
BLOB	Y ₁₅
RECORD	Y
Index, Key, Memo Attributes Supported	
BINARY	Y ₁₃
DUP	Y
NOCASE	Y
OPT	Y
PRIMARY	Y
NAME	Y ₄
Ascending Components	Y
Descending Components	Y
Mixed Components	Y
Field Attributes	Supported
DIM	Y
OVER	Y
NAME	Y
File Procedures	Supported
BOF(<i>file</i>)	Y
BUFFER(<i>file</i>)	N
BUILD(<i>file</i>)	Y

BUILD(<i>key</i>)	Y
BUILD(<i>index</i>)	Y
BUILD(<i>index, components</i>)	Y
BUILD(<i>index, components, filter</i>)	Y
BYTES(<i>file</i>)	Y
CLOSE(<i>file</i>)	Y
COPY(<i>file, new file</i>)	Y
CREATE(<i>file</i>)	Y
DUPLICATE(<i>file</i>)	Y
DUPLICATE(<i>key</i>)	Y
EMPTY(<i>file</i>)	Y
EOF(<i>file</i>)	Y
FLUSH(<i>file</i>)	Y
LOCK(<i>file</i>)	Y ₅
NAME(<i>label</i>)	Y
OPEN(<i>file, access mode</i>)	Y
PACK(<i>file</i>)	Y ₆
POINTER(<i>file</i>)	Y ₈
POINTER(<i>key</i>)	Y ₈
POSITION(<i>file</i>)	Y ₉
POSITION(<i>key</i>)	Y ₉
RECORDS(<i>file</i>)	Y
RECORDS(<i>key</i>)	Y
REMOVE(<i>file</i>)	Y
RENAME(<i>file, new file</i>)	Y
SEND(<i>file, message</i>)	Y
SHARE(<i>file, access mode</i>)	Y

STATUS(<i>file</i>)	Y
STREAM(<i>file</i>)	Y ₇
UNLOCK(<i>file</i>)	Y
Record Access	Supported
ADD(<i>file</i>)	Y
ADD(<i>file</i> , <i>length</i>)	N
APPEND(<i>file</i>)	Y
APPEND(<i>file</i> , <i>length</i>)	N
DELETE(<i>file</i>)	Y ₂
GET(<i>file</i> , <i>key</i>)	Y
GET(<i>file</i> , <i>filepointer</i>)	Y ₈
GET(<i>file</i> , <i>filepointer</i> , <i>length</i>)	N
GET(<i>key</i> , <i>keypointer</i>)	Y
HOLD(<i>file</i>)	Y
NEXT(<i>file</i>)	Y
NOMEMO(<i>file</i>)	Y
PREVIOUS(<i>file</i>)	Y
PUT(<i>file</i>)	Y
PUT(<i>file</i> , <i>filepointer</i>)	Y
PUT(<i>file</i> , <i>filepointer</i> , <i>length</i>)	N
RELEASE(<i>file</i>)	Y
REGET(<i>file</i> , <i>string</i>)	Y
REGET(<i>key</i> , <i>string</i>)	Y
RESET(<i>file</i> , <i>string</i>)	Y
RESET(<i>key</i> , <i>string</i>)	Y
SET(<i>file</i>)	Y
SET(<i>file</i> , <i>key</i>)	Y

SET(<i>file, filepointer</i>)	Y
SET(<i>key</i>)	Y
SET(<i>key, key</i>)	Y
SET(<i>key, keypointer</i>)	Y
SET(<i>key, key, filepointer</i>)	Y
SKIP(<i>file, count</i>)	Y
WATCH(<i>file</i>)	Y
Transaction Processing	Supported (See Note 10)
LOGOUT(<i>timeout, file, ..., file</i>)	Y11
COMMIT	Y
ROLLBACK	Y
Null Data Processing	Supported
NULL(<i>field</i>)	N
SETNULL(<i>field</i>)	N
SETNONNULL(<i>field</i>)	N

Notes

- 1 We recommend using a variable password that is lengthy and contains special characters because this more effectively hides the password value from anyone looking for it. For example, a password like "dd....#\$...*&" is much more difficult to "find" than a password like "SALARY."

Tip

To specify a variable instead of the actual password in the Owner Name field of the File Properties dialog, type an exclamation point (!) followed by the variable name. For example: !MyPassword.

- 2 The TopSpeed driver automatically reclaims space freed by deleted records and keys.
- 3 The TopSpeed file system uses the same compression algorithm for RECORDs and MEMOs. For data of 255 bytes or less, MEMOs have no disk space advantage over STRINGs. However, STRINGs are always allocated space (RAM) within the record buffer, whereas MEMOs are only allocated space when the file is OPENed. MEMOs do carry the advantage of BINARY versus NONBINARY, plus MEMOs may be omitted from all processing with the NOMEMO statement.

- 4 The TopSpeed driver does not support external names for keys, because all keys are stored internally.
- 5 LOCK() only affects other LOCK() calls. The only effect of a successful call to LOCK() is that other processes will get an error FLALLK when they call LOCK().
- 6 PACK performs a BUILD and truncates the file to it's minimum size.
- 7 STREAM has the effect of LOCKing the file.
- 8 GET(*file,filepointer*) requires a pointer value returned from the POINTER() function. POINTER() returns a physical record address (not a record number). Therefore you cannot use

```
GET(file,1)
```

to retrieve the first record in a TopSpeed file because 1 is not a valid pointer in a TopSpeed file.
- 9 POSITION(file) returns a STRING(4). POSITION(key) returns a STRING the size of the key fields + 4 bytes.
- 10 TopSpeed file logging is very fast (about 100 times faster than the Clarion driver). With LOGOUT, the TopSpeed engine posts all transactions to memory. ROLLBACK simply frees the memory, while COMMIT writes out the database changes in a stream.

If a system crashes during a transaction (LOGOUT--COMMIT), the recovery is automatically handled by the TopSpeed driver the next time the affected file is accessed.
- 11 LOGOUT has the effect of LOCKing the file. See also *PROP:Logout* in the *Language Reference*.
- 12 THREADED files do not consume additional file handles for each thread that accesses the file.
- 13 OEM conversion is not applied to BINARY MEMOs and BLOBs.
- 14 The TopSpeed driver accomplishes case insensitivity by converting strings to lowercase. This can cause unexpected behavior for characters that fall between the upper and lower case alphabet (that is, ^ (94) and _ (95) for both ANSI and ASCII sequences).
- 15 The driver can store BLOBs up to 640 MB. If you attempt to store a BLOB bigger than this, an ERRORCODE 80 - Function not supported, is returned. This error is returned after the BLOB handle assignment:
(e.g., blobname{PROP:Handle} = image{PROP:Handle}).

TopSpeed:Other

File Sharing

- † SHARE and open access modes:

The following open access modes are supported *Share required*

34 (12h) Read/Write, deny write (default for OPEN) Yes
66 (42h) Read/Write, deny none (default for SHARE) Yes
64 (40h) Read Only, deny none Yes
18 (12h) Read/Write, deny all No
16 (10h) Read Only, deny all No
32 (20h) Read Only, deny write No

For the modes indicated, SHARE.EXE (which implements DOS record locking) must be loaded in AUTOEXEC.BAT or CONFIG.SYS. The following example loads SHARE in AUTOEXEC.BAT, providing 500 maximum file locks, and the default 2048 bytes for the storage area.

```
C:\DOS\SHARE.EXE /L:500
```

If SHARE.EXE is required but is not loaded when the driver tries to obtain a lock, the program generates a run-time message of "Failed to re-open in exclusive mode."

Tip

You do not need SHARE.EXE (or VSHARE) in any environment (for example, Novell, Win95 or Win NT) that supplies file locking as part of the operating system.

If there is no form of SHARE present (SHARE or VSHARE), then, for the first file access, the driver opens the file in exclusive mode. Thereafter, subsequent attempts to open the file will fail.

ERRORCODE 90

The TopSpeed driver posts an ERRORCODE of 90 for unexpected runtime errors. At the same time, the driver posts a FILEERRORCODE (the former TPSBT error code) that helps us diagnose the problem. This error handling gives you more control over runtime errors and provides us with more information. That is, your program can trap for ERRORCODE=90 and react accordingly.

Should you receive an ERRORCODE of 90 from the TopSpeed driver, we want to know about it. Please send us a copy of the file and the corresponding FILEERRORCODE value.

Large Keys (or small RAM)

APPEND() is recommended over ADD() if the total size of the keys exceeds the amount of RAM available, if there is more than one key, or when adding a large number of records. The size of a key (for this purpose) is the number of entries times (the sum of key fields + 10 bytes). If the records being added are already in an approximate key order, then you can discount that key for the purposes of the above calculation.

As an example, if a file has two 40 byte keys and 2 Megabytes of RAM are available, then ADD() becomes (relatively) slow when the database size exceeds about $2,000,000 / (40 + 10 + 40 + 10) = 20,000$ records.

Incremental Key/Index Build

The TopSpeed driver implements incremental building; this means that building a key only reads records starting from the first record appended since the key was last built. The driver merges the new keys with the existing key. Thus building a large key where only a few recently added records have been modified should be *fast*. See the FULLBUILD driver string above.

Building an index is similar, but must start at the minimum physical record whose position in the index has changed since the index was last built.

Dynamic indexes are not retained, so cannot be built incrementally.

Batch Processing Performance

When *writing* a large number of records, use STREAM() or open the file in a deny write mode, that is, OPEN(file) rather than SHARE(file). After the records have been written, call FLUSH() to allow other users access.

It is very important to use STREAM() when ADDing/APPENDing/PUTting a large number of records. STREAM() will typically make processing about 20 times faster. For example, adding 1000 records might take nearly 2 minutes without STREAM(), but *only 5 seconds* with STREAM.

It is not necessary to use STREAM() or FLUSH() on a logged out file (performance on logged out files is always good).

STREAM has the effect of LOCKing the file.

POINTERS and Deleted Records

POINTER(key) returns the relative position of the record within the file. Consequently when that record is DELETED, the pointer becomes invalid. Any subsequent access using the pointer fails. If you require fuzzy matching whereby the nearest record is returned, use the POSITION() function.

Data Compression--STRINGS vs MEMOs

The TopSpeed driver compresses the entire record buffer area (not individual fields within the record), therefore, compression gains can be realized by placing similar fields adjacent to each other in the FILE declaration.

The TopSpeed file system uses the same compression algorithm for STRINGS and MEMOs; however, the compression occurs at a "higher level" for MEMOs than for STRINGS. As a result, MEMOs do have a disk space advantage over large STRINGS (over 500 bytes) and smaller STRINGS can have a slight performance advantage over MEMOs. The larger the STRING, the greater the advantage.

MEMOs do carry the advantage of BINARY versus NONBINARY, plus MEMOs may be omitted from all processing with the NOMEMO statement.

STRINGS are always allocated space (RAM) within the record buffer, whereas MEMOs are only allocated space when the file is OPENed. Also MEMOs cannot be key components.

Estimating File Size

The TopSpeed file driver compresses data and key information, so the ultimate file size depends on the "compressibility" of the data and keys. In the worst case (data and keys cannot be compressed because there is no repeating information) the file size may be estimated as:

`(RecordSize + All Key components) * Records + Fixed Overhead`

In a more realistic case (data and keys are compressible), the file size may be estimated as:

`((size of all string fields)/(compressibility factor) + size of all binary fields + size of all binary key components + (4 * number of string key components)) * Records + Fixed Overhead`

Note that Fixed Overhead varies depending on your file definition. Fixed overhead includes about 800 bytes for the driver, plus the header information describing the fields and keys for the file. The more fields and keys, and the longer the names, the higher the fixed overhead. A rough rule of thumb for calculating fixed overhead is 800 bytes + 40 bytes for each field and key. For Example:

File Description	Estimated Fixed Overhead
1 field, no keys	1KB
20 fields, 10 keys	2KB
200 fields, 10 keys	9KB

Concurrent User Limit

The TopSpeed driver limits concurrent users to 1024 per file; additional users would have to wait momentarily until a slot opens up. Practically speaking the driver is very unlikely to reach this limit since very few networks and servers will support this many concurrent users. Generally, we recommend a client/server file system for more than 30 concurrent users.

Transaction Processing--the TCF File

Speedy Logging and Automatic Recovery

TopSpeed transaction logging is very fast (about 100 times faster than the Clarion driver). With LOGOUT, the TopSpeed engine posts all transactions to memory. ROLLBACK simply frees the memory, while COMMIT writes out the database changes in a stream.

If a system crashes during a transaction (LOGOUT--COMMIT), the recovery is automatically handled by the TopSpeed driver the next time the affected file is accessed.

The Transaction Control File

The transaction control file (.TCF) is used to ensure that changes to more than one DOS file, which are grouped into a transaction, either all happen or none happen. By default the transaction control file has the name "\TOPSPEED.TCF." The TCF driver string lets you change this. See *TCF* for more information.

When any workstation finds a file which is in a partially committed state, and which was involved in a multi-file transaction, it needs to access the TCF file to decide what to do. The TCF file provides "atomicity"--a single (boolean) storage location which indicates if a multi-file transaction committed or not.

Note that the .TCF file contains very little information; it just serves to coordinate multi-file commits. The actual rollback/commit data is stored in the data (.TPS) files.

How TopSpeed Transaction Logging Works

LOGOUT gives each transaction a unique id which it stores in the .TCF file. LOGOUT also stores the .TCF file name and transaction id in each data (.TPS) file which is updated, so that after a crash, the next time the file is opened the TopSpeed driver can find the .TCF file and do any necessary recovery. COMMIT removes the unique transaction id from the .TCF file.

To be effective, the .TCF file must be accessible when any files controlled by it are accessed. Therefore, you generally should not delete or move .TCF files. If a transaction updates network files, you should specify a transaction control file on the network.

It is not necessary to use the same TCF file for all transactions; however, we strongly recommend it. The consequence of there being several TCF files with various levels of accessibility (or of a deleted or overwritten TCF file) is that some of the files within a transaction might be updated and others left unchanged.

Storing Multiple Tables in a single .TPS File

By using the characters '\!' in the NAME() attribute of a TopSpeed file declaration, you can specify that a single .TPS file will hold more than one table. For example, to declare a single .TPS file 's&p.tps' that contains 3 tables called *supp*, *part* and *ship*:

```
Supp FILE,DRIVER('TopSpeed'),PRE(Supp),CREATE,NAME('S&P\!Supp')
...
Part FILE,DRIVER('TopSpeed'),PRE(Part),CREATE,NAME('S&P\!Part')
...
Ship FILE,DRIVER('TopSpeed'),PRE(Ship),CREATE,NAME('S&P\!Ship')
...
```

The tables share a single DOS file handle, opened when the first table is opened, and closed when the last table is closed. The first open mode determines the open mode for *all* the other tables in the file. If the first open mode is read-only, then all the tables are read-only and no updates are allowed.

Similarly, if one of the tables in the file is logged out, then all the tables are effectively logged out. If one table in the file is flushed, then all the tables are flushed.

This feature is especially useful when there are a large number of small tables, or when the application must normally access several related tables at once.

You can retrieve the names of tables within the .TPS files with the SEND() command. To retrieve the first name, issue:

```
SEND(file,'PNM=')
```

This returns the name of the first table. To retrieve the second name, issue:

```
SEND(file,'PNM=FirstTableName')
```

This returns the name of the second table, and so on.

You can also rename the tables; for example, given the above declarations the following command renames the table called Supp to Old_Supp:

```
RENAME(Supp, 'S&P\!Old_Supp')
```

If you use the OWNER attribute on multiple tables in a single .TPS file, all the tables must have the same OWNER attribute.

If you don't specify a table name, the table is called 'unnamed', so that the following are all equivalent:

```
foo    FILE,DRIVER('TopSpeed')
foo    FILE,DRIVER('TopSpeed'),NAME('foo')
foo    FILE,DRIVER('TopSpeed'),NAME('foo\!unnamed')
```

Collating Sequences

Changing Collating Sequence

Changing the collation sequence on a Clarion 2.003 or earlier TopSpeed file (by changing .ENV file or OEM flag) corrupts the file.

This is no longer true, because the collating sequence for the file is now stored within the file. This change is fully backward compatible. Old files continue to work as before and new files are accessible by older programs.

To add the collating sequence information to an existing file, simply do a full build on the file:

```
SEND(file, 'FULLBUILD=on')
BUILD(file)
```

The collating sequence for a TopSpeed file is established when the table is created or a full build is performed. Therefore the OEM flag is only significant at the creation of the file or on a full build.

Any application that uses an incorrect sequence (due to an incompatible .ENV file) to access a file may get unpredictable results, but will not corrupt the data.

Accessing TopSpeed files with Access Jet and ODBC

Occasionally, the Access Jet engine returns "#deleted" for each requested field. This is a known bug in Microsoft Access. The default query used by Access does an internal comparison of the record set to determine if a record has been deleted or modified from the database. The mechanism is known to work poorly for certain data types, notably DECIMAL.

Microsoft recommends using an SQL pass-through query as a work around to this problem. To create a SQL pass-through query:

1. Choose SQL Specific, Pass Through from the Query menu in query design mode. For Access 7, Select Query, and press the New button.
2. Accept the default of Design View.
3. Close the Show Table Window.
4. Select SQL Specific from the Query menu and select the Pass-Through option.
5. Enter the SQL statement.

The Query can be saved for future use. This method will correct virtually all display problems, but the resulting grid is not updatable. Updates must be performed using an Update Query when SQL Passthrough is used.

TopSpeed Database Recovery Utility

The TopSpeed file system is designed to automatically repair most errors. However, if a TopSpeed file is physically damaged during a system malfunction, the TopSpeed Database Recovery Utility can recover the undamaged portions of your data.

Note:

The TopSpeed Database Recovery Utility is an emergency repair tool and should not be used on a regular basis. Use it only when a file has been damaged.

The TopSpeed Database Recovery Utility reads the damaged file and writes the recovered records to a new file. It uses the information stored in the file's header and scans the file recovering undamaged portions.

Optionally, you can provide an example file containing the header information in the event the original header information is damaged. An example file is any file with a FILE declaration identical to the damaged file. You can create an example file by issuing a CREATE(file) command, then saving the resulting empty file to a new name.

The TopSpeed Database Recovery Utility is a distributable utility designed to help your end users recover damaged files.

Tip

The Clarion license agreement applies to TPSFIX.EXE. You may distribute to your users, but they may not redistribute it.

The recovery utility is designed to work either interactively or noninteractively with command line parameters. Interactively, you provide the parameters through two wizard dialogs. You can run TPSFIX noninteractively by supplying the command line parameters with the Clarion RUN() statement, Windows API calls, Windows 95 shortcuts, or Program Manager Icons.

ERRORCODE 90 and Corrupted Files

The TopSpeed driver posts an ERRORCODE of 90 for unexpected runtime errors. When an ERRORCODE of 90 occurs, the driver also posts a FILEERRORCODE (the former TPSBT error code) that helps us diagnose the problem.

An ERRORCODE of 90 usually indicates your TopSpeed file is corrupted. In most cases the corruption is a result of hardware failure. For example, one customer with a 50 machine network traced a near daily file corruption to bad network cards on 2 of the 50 machines. After replacing the bad cards, the corruptions disappeared.

However, should you receive an ERRORCODE of 90 from the TopSpeed driver, we want to know about it. Before you repair the file, please make a copy of the damaged file and send it to us along with the corresponding FILEERRORCODE value. We analyze all the corrupted files we receive for recognizable patterns that can help us improve the driver.

TPSFIX Command Line Parameters

The TPSFIX utility can accept command line parameters which lets you execute it from an application, from a Program Manager Icon, or from a Windows 95 Shortcut.

Here is the syntax for running TPSFIX with command line parameters.

```
TPSFIX sourcepath[?password] [destpath[?password]] [/E:examplepath[?password]]
          [/L:localepath] [/H] [/K] [/P] [/O] [/T:filename]
```

TPSFIX	The executable (TPSFIX.EXE).
<i>sourcepath</i>	The file name and path of the source (damaged) database file.
<i>?password</i>	The file's password.
<i>destpath</i>	The file name and path of the recovered database file. If omitted, the <i>destpath</i> is the same as the <i>sourcepath</i> and an example file is required.
<i>/A</i>	If specified, the user is not offered a backup prompt. The prompt suppressed, however a backup of the file is made.
<i>/E:examplepath</i>	The file name and path of the example database file. This parameter is required for any fix-in-place operation (that is, when <i>sourcepath</i> = <i>destpath</i>).
<i>/H-</i>	If specified, the utility uses the header information in the source file.
<i>/K</i>	If specified, the utility rebuilds all keys for the database.
<i>/L:localepath</i>	The file name and path of the Locale (.ENV) file used to specify an alternate collating sequence.
<i>/N</i>	If specified, the file will be checked for errors. No errors will be corrected.
<i>/O</i>	If specified, the file uses OEMTOANSI and ANSITOOEM to determine the collating sequence. See <i>Internationalization</i> in the <i>Language Reference</i> .
<i>/P</i>	If specified, the user is prompted for each parameter even if they are supplied on the command line.
<i>/T:filename</i>	If there are file errors, a log file with the supplied <i>filename</i> will be created.

Using the Recovery Utility Non-Interactively

There are some issues to consider before running the TPSFIX utility. Because of the following, we do not recommend running TPSFIX from your application program. Rather, it is better to instruct your end users to close down the application program completely before running the TPSFIX utility.

- The database file should NOT be open when running TPSFIX. Ensure the file is closed before starting TPSFIX.

- To prevent access during the recovery process is completed, TPSFIX LOCKs the file automatically.
- It is more efficient and safer to have your application rebuild the KEYS (omit the /K parameter). It is also a good way to check the status of a recovery.

Automatic Fix-in-Place Recovery

By omitting the *destpath* parameter and supplying an example file, you can directly overwrite the damaged file. This is a fix-in-place recovery. The TPSFIX utility does create an intermediate file, but you don't have to worry about it. For Example:

```
TPSFIX.EXE Datafile.TPS /E:Example.TPE /H
```

or with Embedded Source Code:

```
RUN('TPSFIX.EXE Datafile.TPS /E:Example.TPE /H')
```

This recovers the "datafile.TPS" file using the "Example.TPE" file as an example for the table and key layouts, does not rebuild the keys, and uses the header information in the original file. TPSFIX automatically saves the original file to a backup with a file extension of TP1 through TP9. Each time the utility is executed, the numeric portion of the extension is incremented.

Separate Source and Target Recovery

This method requires two lines of embedded source code but gives you control over the renaming process. You insert the source code in the Accepted Embed point for the Menu Item or button. For example:

```
COPY(datafilelabel, 'Datafile.OLD')    ! copies the original file
! to Datafile.OLD
RUN(TPSFIX Datafile.OLD Datafile.tps /H) ! Runs the utility
using the
! renamed file as
! the source and the original
! name as the target
```

This copies the datafilelabel file to DATAFILE.OLD, recovers the file and writes it to DATAFILE.TPS using the header information in the original file.

All SQL Accelerators (Drivers)

General Information for all SQL Drivers

These SQL Accelerator Drivers share a common code base and many common features such as the unique, high speed buffering technology (see *BUFFER* in the *Language Reference*), common driver strings, and SQL logging capability. However, their primary purpose is to translate Clarion file commands into appropriate, efficient SQL statements specific to their respective SQL servers, and to handle any result sets returned by those servers.

The SQL Accelerator Drivers convert standard Clarion file I/O statements and function calls into optimized SQL statements, which they send to their backend SQL servers for processing. This means you can use the same Clarion code to access both SQL tables and other file systems such as TopSpeed files. It also means you can use Clarion template generated code with your SQL databases.

In addition to the automatically generated SQL statements, the SQL Accelerator Drivers forward any additional SQL statements you specify to the backend SQL servers. The SQL Accelerator Driver interprets the result set returned from the SQL server and makes it available to your application program with the Clarion NEXT or PREVIOUS statement.

All the common behavior of all the SQL Accelerator drivers is documented in this chapter. Driver-specific behavior is documented the chapter for that specific SQL driver.

SQL Accelerator Unique Keys

The SQL Accelerator drivers should generally be used only on tables with unique keys. The drivers will function on files without unique keys, but only with substantially limited capabilities. Without a unique key, the RESET and REGET commands return errors, and the driver cannot update the SQL database.

Most Clarion templates also require that you define a primary key for each table in order to generate code.

Using SQL Tables in your Clarion Application

Register the SQL Accelerator Driver

Before your application can use a particular database driver, the driver must be registered with the Clarion development environment. The in-the-box drivers are already registered when you install Clarion. You must register any add-on drivers. See *Clarion's Development Environment--Database Driver Registry* in the *User's Guide* for information on registering database drivers.

Import the Table Definitions

Typically, you add SQL support to your application by importing the SQL table, view, and synonym definitions into your Clarion Data Dictionary. See *The Dictionary Editor--Importing File Definitions* in the *User's Guide* for general information on importing table, file, and view definitions. This section describes SQL Driver imports generally. Driver-specific import information is described in the chapter or manual for each driver.

Although you can manually add table definitions to the dictionary (or even hand code your FILE declarations) for your SQL tables, we strongly recommend importing the table definitions. Importing the table definitions reduces the chance of introducing errors into the dictionary and guarantees the correct specification of data types, key structures, etc.

The importing approach assumes your SQL tables are already defined within the SQL database. In the case where you are designing a new SQL database, you may, of course, lay out the table definitions for the first time in the Clarion Data Dictionary. However, we recommend this approach only for prototyping and for databases with minimum complexity and maintenance requirements. In most cases, to correctly implement an SQL database requires defining more items than are stored in the Clarion Data Dictionary--for example, stored procedures, triggers, access rights, and storage allocation.

Once your table definitions are in the Clarion Data Dictionary, you develop your SQL based applications just as you would any other application.

Note:

Driver-specific import information is described in the chapter or manual for each driver.

SQL Import Wizard--Login Dialog

When you select an SQL Accelerator Driver from the driver drop-down list, the Import Wizard opens the **Login/Connection** dialog. The **Login/Connection** dialog collects the connection information for the SQL database.

Note:

Before you can connect to the SQL database and import table definitions, the database must be started and must be accessible from your computer.

Fill in the fields in the **Login/Connection** dialog.

Next >

Press this button to open the Import Wizard's **Import List** dialog.

SQL Import Wizard--Import List Dialog

When you press the **Next >** button, the Import Wizard opens the **Import List** dialog. The **Import List** dialog lists the importable items.

Highlight the table, view, or synonym whose definition to import, then press the **Finish** button to import. The Import Wizard adds the definition to your Clarion Data Dictionary, then opens the **File Properties** dialog to let you modify the default definition.

Import additional tables, views, and synonyms by repeating these steps. After all the items are imported, return to the Dictionary Editor where you can define relationships and delete any columns not used in your Clarion application. See *Advanced Techniques--Define Only the Fields You Use*.

Connection Information and Driver Configuration--File Properties

Typically, you add SQL support to your application by importing the SQL or ODBC table, view, and synonym definitions into your Clarion Data Dictionary. The Import Wizard automatically fills in the **File Properties** dialog with default values based on the imported item. However, there are several fields in the **File Properties** dialog you can use to further configure the way the SQL Accelerator Driver accesses the data. These **File Properties** fields are described below.

Driver Options

Typically, the Import Wizard places nothing in the **Driver Options** field. However, you can add driver strings to this field to control how the driver accesses your SQL data. For example, you can generate a log of driver activity or specify how the driver handles dates with a value of zero (0). See *SQL Driver Strings* for more information.

Owner Name

Typically, the Import Wizard places the SQL database connection information (Host, Username, Password, etc.) in the **Owner Name** field.

For security and portability reasons, you may want to specify this connection information with variables rather than hard coded strings in your dictionary. To use a variable specification, type the variable name, preceded by an exclamation point in the **Owner Name** field; for example !LoginString. Then use whatever method you choose to prime the variable before accessing the SQL table.

Some SQL Accelerator drivers allow additional information in the **Owner Name** field. This information is described in the documentation for each driver.

SQL Driver Behavior

Automatic Login Dialog

The SQL Accelerator drivers automatically look for Username and Password values whenever they access an SQL table. If a Username and Password have already been supplied, the driver uses those values. If no values have been supplied, the driver prompts for the Username and Password with the automatic login dialog.

We recommend opening a table at the start of your program so the time devoted to logging in occurs at program start up. Clarion's Application Wizard automatically generates code to do this for SQL Accelerator drivers. However, if you do not use the Application Wizard, you can accomplish the same effect simply by adding an SQL table to the File Schematic for your main procedure. This automatically generates code to open the table.

Except for the ODBC Accelerator Driver, the automatic Login dialog lets the user specify Username, Password and Database.

In the **Database** drop-down list, select from previously selected hosts. If the **Database** list is empty, you may type in the database name.

SET/NEXT and SET/PREVIOUS Processing (SELECT/ORDER BY)

A SET statement followed by a NEXT in a LOOP structure is the most common Clarion method to process records sequentially. When the SQL Accelerator drivers encounter a SET/NEXT combination, they generate an SQL SELECT statement with an ORDER BY clause based on the KEY component fields. The KEY component fields are determined by the KEY names in the SET statement. For example, the SQL driver translates this Clarion code

```

Ord      FILE,PRE(Ord),DRIVER('SQLDriver'),NAME('ord')
NameDate KEY(+Ord:Name,+Ord:Date),NAME('DateKey')
Record   RECORD
Name     STRING(12),NAME('NameId')
Date     DATE,NAME('OrderDate')
Type     STRING(1),NAME('OrderType')
Details  STRING(20),NAME('OrderDetails')
        END
        END
CODE
Ord:Name = 'SMITH'
SET(Ord:NameDate,Ord:NameDate)
LOOP
  NEXT(Ord)
  !... some processing
END

```

into a SELECT statement similar to:

```
SELECT NameId,OrderDate,OrderType,OrderDetails FROM Ord
WHERE (NameID >= 'SMITH')
ORDER BY NameID, OrderDate
```

Tip

The SET(*file*) statement (to process in file order, not keyed order) only supports the NEXT statement. Any attempt to execute a PREVIOUS statement when processing in file order causes ERRORCODE 80 (Function Not Supported).

NULL Fields

When you read a row with NULL values from an SQL table, the Clarion record buffer contains an empty string for string fields, or a 0 for numeric fields, and NULL(field) returns TRUE for the field. If the field's contents are later changed to a non-empty or non-zero value then NULL(field) returns FALSE.

If you want to change a NULL field to non-null, but still blank or zero, then you must call SETNONULL(field) to reset the null flag.

If you wish to clear a field to NULL that was previously non-null then call SETNULL(field) or SETNULL(record). SETNULL() clears the contents of the field or record and resets the null flag.

When adding a new record to a file, by default all blank fields are added as blank or zero fields, not as NULL. If you want to force a field to be added with a NULL value, then you must call SETNULL(field) or SETNULL(record) to null all the fields.

Performance Considerations

Generally, Clarion's development environment (Data Dictionary Import Wizard, Database Drivers, and templates) produces optimized, high performance, SQL applications.

This section describes some of the issues involved in producing these optimized applications. You should be aware of these issues so you can maintain a high level of performance as you take more control of the development process.

Define Only the Fields You Use

With the SQL Accelerator drivers you only need to define the fields that you actually use in the Clarion Data Dictionary. This reduces both the overhead within your Clarion application and network traffic.

For example, if your SQL table contains 200 columns but only three are needed for a particular program, retrieving only those three fields dramatically reduces the amount of data sent over the network. If each column contains 20 bytes, then three columns would require only 60 bytes to be transferred whereas all 200 columns requires a 4,000 byte transfer.

After you have imported the table definition into your Clarion Data Dictionary, use the Dictionary Editor's **Columns / Key Definition** dialog to delete the fields/columns you don't use.

Matching Clarion Keys to SQL Constraints and Indexes

Generally, the Clarion KEY definition need not exactly match an index in the SQL database. The Clarion KEY simply serves to supply the appropriate ORDER BY clause for driver generated SELECT statements.

However, if the Clarion KEY does not match an SQL key or index, then the SQL server must build a temporary logical view every time you access the table using the unmatched KEY. This can be very slow for large files.

The best way to guarantee the Clarion KEYs have a matching SQL constraint or index, is to import the table, view, or synonym definition into the Clarion Data Dictionary. See *Import the Table Definitions*.

Filter (Contracting) Locators

Using Filter Locators on your BrowseBox controls rather than Incremental or Step Locators can reduce the volume of data sent between client and server. See *BrowseBox Control Template* for more information on Filter Locators.

Approximate Record Count

By default, the Clarion templates generate code to count the total number of records to be processed for a report. This total record count allows for an accurate progress bar display during report generation. However, for large tables, the resulting SELECT COUNT(*) can be very slow.

Therefore, for large reports, we recommend providing an approximate record count to suppress the SELECT COUNT(*) as follows:

1. In the **Application Tree** dialog, RIGHT-CLICK the (*Report*) procedure, then choose **Properties** from the popup menu.
This opens the **Procedure Properties** dialog.
2. Press the **Report Properties** button to open the **Report Properties** dialog.
3. In the **Approx. Record Count** field, type an approximate record count for the report, such as *5000*.
4. Press the **OK** button to close the **Report Properties** dialog.
5. Press the **OK** button again to return to the **Application Tree** dialog.
6. Press the button to save your work.

Fixed Thumbs and Movable Thumbs

By default, Clarion's code generation Wizards use Fixed Thumbs when Browsing SQL tables because Movable Thumbs can cause major performance slow downs on large tables in Clarion / SQL applications. For this reason, we recommend that you specify Fixed Thumbs for your manually place BrowseBox controls as follows:

1. In the **Application Tree** dialog, RIGHT-CLICK the Browse procedure, then choose **Extensions** from the popup menu.
This opens the **Extension and Control Templates** dialog.
2. In the list box, select *Browse on ...*, then press the **Scroll Bar Behavior** button.
This opens the **Scroll Bar Behavior** dialog.
3. In the **Scroll Bar Type** drop-down list, select *Fixed Thumb*, then press the **OK** button.
4. Press the **OK** button again to return to the **Application Tree** dialog.
5. Press the **Save** button  to save your work.

Date and Time Column Considerations

A common practice in some SQL databases (MSSQL, Oracle, and others) is to define a composite DateTime column (i.e., one column representing two pieces of data). In order for Clarion to separate the date and time information for processing, an import of this DateTime column type into the Dictionary Editor results in the following type of data structure:

```
Orderdate  STRING(8)           !original column name from SQL
Orderdate_GROUP  GROUP, OVER(Orderdate) !structure created by Clarion
Orderdate_DATE    DATE         !use this column to reference date info
Orderdate_TIME    TIME         !use this column to ref time info
END
```

No matter what type of SQL/ADO/ODBC driver you are using, Clarion will detect and convert these composite DateTime columns for you automatically.

Know your back end! For example, the SMALLDATETIME and DATETIME data types of MS-SQL are treated equally, with both only being able to store the minimum of either the precision of the Clarion TIME field or the backend data type. So, in the case of the SMALLDATETIME data type, the seconds and hundredths of a second are discarded, using the SMALLDATETIME rule that > 29.99 is rounded up to the next minute.

Another note; If you don't need the time portion, you can just use a "Date" type field (as long as no one else is writing to the "Time" portion from another application). Otherwise, you will need to be aware that the time portion is not zero when filtering, sorting, etc.

SQL Batch Transaction Processing

Most SQL databases operate in auto-commit mode. This means that any operation that updates a table (ADD, PUT, or DELETE) executes an implicit COMMIT. This can be very slow for a series (batch) of updates.

To optimize batch processes, surround any batch processing in a transaction frame (that is, with LOGOUT and COMMIT). The LOGOUT command prevents any subsequent implicit COMMITs until the transaction frame ends with either a COMMIT or a ROLLBACK. For example:

```

LOGOUT(.1,OrderDetail)           !Begin Transaction
DO 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               !check for errors
  Det:Record = DetailQue        !Assign to record buffer
  ADD(OrderDetail)              !and add it to the file
  DO ErrorHandler               !check for errors
END
COMMIT                           !Terminate good transaction

ErrorHandler ROUTINE            !Error routine
IF NOT ERRORCODE() THEN EXIT.   !Bail out if no error
ROLLBACK                         !Rollback the bad transaction
MESSAGE('Transaction Error - ' & ERROR())!Log the error
RETURN                            !and get out

```

You may want to issue intermittent calls to COMMIT and LOGOUT to save data at regular intervals. See the *Language Reference* for more information.

Using Embedded SQL

You can use Clarion's property syntax (PROP:SQL) to send SQL statements to the backend SQL server, within the normal execution of your program. For backward compatibility, you can also use the SEND function to send SQL statements; however, we recommend using the property syntax.

Note: When you issue a SELECT statement using PROP:SQL, the selected fields must match the fields declared in the named file or view. **In addition, if you use VIEW{Prop:SQL} to issue a SELECT statement, the fields in the SELECT must be ordered based on the field order in the file definition, not the PROJECT sequence.**

PROP:SQL

You can use Clarion's property syntax (PROP:SQL) to send SQL statements to the backend SQL server, within the normal execution of your program. You can send any SQL statements supported by the SQL server.

This capability lets your program do backend operations independent of the SQL Accelerator driver's generated SQL. For example, multi-record updates can often be accomplished more efficiently with a single SQL statement than with a template generated Process procedure which updates one record at a time. In cases like these it makes sense for you to take control and send custom SQL statements to the backend, and PROP:SQL lets you do this.

If you issue an SQL statement that returns a result set (such as an SQL SELECT statement), you use NEXT(file) to retrieve the result set one row at a time, into the file's record buffer. The FILEERRORCODE() and FILEERROR() functions return any error code and error message set by the back-end SQL server.

You may also query the contents of PROP:SQL to get the last SQL statement issued by the file driver.

Examples:

```
SQLFile{PROP:SQL}='SELECT field1,field2 FROM table1'      |
                  & 'WHERE field1 > (SELECT max(field1)' |
                  & 'FROM table2'                       |
                                                         |Returns a result set you
                                                         | get one row at a time
                                                         | using NEXT(SQLFile)

SQLFile{PROP:SQL}='CALL GetRowsBetween(2,8)'            |!Call stored procedure

SQLFile{PROP:SQL}='CREATE INDEX ON table1(field1 DESC)' |!No result set

SQLFile{PROP:SQL}='GRANT SELECT ON mytable TO fred'    |DBA tasks

SQLString=SQLFile{PROP:SQL}                            |Get last SQL statement
```

SEND and PROP:SQL

You can use the Clarion SEND procedure to send an SQL command to the backend database. This is provided for backward compatibility with early versions of Clarion. We recommend using the property syntax to send SQL statements to the backend database.

Examples:

```
SEND(SQLFile,'SELECT field1,field2 FROM table1'      |
      & 'WHERE field1 > (SELECT max(field1)'        |
      & 'FROM table2')          !Returns a result set you
                                ! get one row at a time
                                ! using NEXT(SQLFile)

SEND(SQLFile,'CALL GetRowsBetween(2,8)')    !Call stored procedure

SEND(SQLFile,'CREATE INDEX ON table1(field1 DESC)') !No result set
```

Using Embedded SQL for Batch Updates

SQL does a good job of handling batch processing procedures such as: printing reports, displaying a screen full of table rows, or updating a group of table rows.

The SQL Accelerator drivers take full advantage of this when browsing a table or printing. However, they do not always use it to its best advantage with the Process template or in code which loops through a table to update multiple records. Therefore, when doing batch updates to a table, it can be much more efficient to execute an embedded SQL statement than to rely on the code generated by the Process template.

For example, to use PROP:SQL to increase all Salesman salaries by 10% you could:

```
SQLFile  FILE,DRIVER('Oracle'),NAME(SalaryFile)
Record   RECORD
SalaryAmount  PDECIMAL(5,2),NAME('JOB')
.
.
CODE
sqlFile{PROP:SQL} = 'UPDATE SalaryFile SET '&|
                    'SALARY=SALARY * 1.1 WHERE JOB='S'''
```

The names used in the SQL statement are the SQL table names, not the Clarion field names.

PROP:SQLFilter

You can use PROP:SQLFilter to filter your VIEWS using native SQL code rather than Clarion code.

When you use PROP:SQLFilter, the SQL filter is passed directly to the server. As such it cannot contain the name of variables or functions that the server is not aware of; that is the filter expression must be valid SQL syntax with valid SQL column names. For example:

```
View{PROP:SQLFilter} = 'Date = TO_DATE(''01-MAY-1996'', 'DD-MON-YYYY')'
```

or

```
View{PROP:SQLFilter} = 'StrField LIKE ''AD%'''
```

Combining VIEW Filters and SQL Filters

When you use PROP:SQLFilter, the SQL filter may replace any filter specified for the VIEW, or it may be in addition to a filter specified for the VIEW. Prefix the SQL filter with a plus sign (+) to append the SQL filter to the VIEW filter specified. For example:

```
View{PROP:SQLFilter} = '+ StrField LIKE ''AD%'''
```

When you append the SQL filter by using the plus sign, the logical end result of the filtering process is (View Filter) AND (SQL Filter).

Omit the plus sign (+) to replace the Clarion filter with the SQL filter. When you replace the Clarion filter with the SQL filter by omitting the plus sign, the logical end result of the filtering process is simply (SQL Filter).

Calling a Stored Procedure

To call a stored procedure the following SQL syntax is used to build the SQL calling statements.

[output_bound_field =] call_type [[parameter[,parameter]...]]

<i>call_type</i>	CALL NORESLTCALL
<i>parameter</i>	constant bound_field
<i>constant</i>	This must conform to the syntax of your backend. Normally numerics and strings are the same as Clarion. For ODBC systems, date constants are in the format {d 'yyyy-mm-dd'}, time constants are {t 'hh:mm:ss'} and time stamp constants are {ts 'yyy-mm-dd hh:mm:ss'}.
<i>bound_field</i>	output_bound_field output_bound_field '['bind_type']'
<i>output_bound_field</i>	&variable
<i>bind_type</i>	IN OUT INOUT BINARY (valid on all SQL drivers, except Oracle)
<i>variable</i>	This must be a variable that you have previously bound using the BIND function.

CALL

To call a stored procedure you use property syntax to issue the SQL syntax 'CALL *storedprocedure*.'

Example:

```
MyFile{PROP:SQL} = 'CALL SelectRecordsProcedure (&MyVar[INOUT])'
```

NORESULTCALL

The SQL Accelerator drivers also allow the syntax 'NORESULTCALL *storedprocedure*' for stored procedures that do not return a result set.

Example:

```
MyFile{PROP:SQL} = 'NORESULTCALL SelectRecordsProcedure (&MyVar[INOUT])'
```

Return Values

The Accelerator drivers support return codes, output parameters, and in/out parameters for stored procedures. These are defined using IN, OUT, and INOUT. IN declares a variable as input, OUT declares a variable as output, and INOUT declares a variable as both input and output. You can also have your stored procedures return a result set.

The BINARY switch is used to signal the driver to pass the data in the bound field as binary data rather than character data. See the example below.

Example:

```
MyFile FILE,DRIVER('ODBC')
Record    RECORD
ErrorCode  LONG
ErrorMsg   STRING(100)
          END
          END
          CODE
OPEN(MyFile)
MyFile{PROP:SQL} = 'CALL ProcWithResultSet'
NEXT(MyFile)
IF ~ERRORCODE()
  IF MyFile.ErrorCode THEN STOP(MyFileErrorMsg).
END
```

Note: The above example shows how to return a result set. The result set must match the fields declared in the named file or view. The storedprocedure *ProcWithResultSet* includes a final select statement that results with the set of requested data.

Example:

```

PROGRAM
MAP
  CallProc (STRING)
END

MyFile  FILE,DRIVER('MSSQL')
Record  RECORD
c        LONG
        END
        END

Ret      LONG
Out      STRING(10)

CODE

BIND('RetCode', Ret)
BIND('Out', Out)
CallProc('&RetCode = CALL StoredProcTest(''1'',&Out)')
MESSAGE(Return value of StoredProcTest = ' & Ret)
MESSAGE(Output parameter of StoredProcTest = ' & Out)

CallProc PROCEDURE(Str)
CODE
MyFile{PROP:SQL} = Str

```

Note:

The above example shows how to return an output parameter.

Example:

```

PROGRAM

MAP
END

PRAGMA('link(C%V%MSS%X%%L%.LIB)')

SQLFile FILE,DRIVER('MSSQL'),NAME('SYSFILES')
REC      RECORD
ID       LONG
NAME     CSTRING(100)
        END
        END

```

```

TS          STRING(8)
CODE
  OPEN(SQLFile)
  SQLFile{PROP:SQL} = 'DROP PROCEDURE tstest'
  SQLFile{PROP:SQL} = 'CREATE PROCEDURE tstest @ts as timestamp AS '& |
                      ' return'

  BIND('TS',TS)
  TS='<0><0><0><0><0><0><5H><0DDH>'
  SQLFile{PROP:SQL}='NORESETCALL TSTEST(&TS[IN][BINARY])'

```

Note:

The above example shows how to use the IN and BINARY switches.

For a more specific example tailored to MSSQL, refer to the MSSQL Accelerator topic.

Runtime SQL Properties for Views using SQL Drivers

The SQL View Engine allows you to specify SQL that will be substituted for a column in a SELECT statement using the following syntax:

```
view{'field_label',PROP:Name} = SQLString
```

where *SQLString* is any SQL valid within a SELECT statement.

Example:

```

PROGRAM
MAP
END

EMP      FILE,DRIVER('ORACLE'),NAME('EMP'),PRE(EMP)
P_EKY_EMP  KEY(EMP:EMPNO),NOCASE,OPT,PRIMARY
KEY_DEP   KEY(EMP:DEPTNO),DUP,NOCASE,OPT
Record    RECORD
EMPNO     SHORT          !Emp-no
ENAME     CSTRING(11)    !Employee name
JOB       CSTRING(10)    !Job
HIREDATE  DATE           !Hiredate
MGR       SHORT          !Manager
SAL       PDECIMAL(7,2)  !Salary
COMM     PDECIMAL(7,2)  !Commisison
DEPTNO    BYTE
          END
          END

MyView VIEW(EMP)
        PROJECT(EMP:EmpNo)
        END

```

```
CODE
OPEN(EMP)
OPEN(MyView)
MyView{'EMP:EmpNo',PROP:NAME} = 'count(*)'
SET(MyView)
NEXT(MyView)
```

This example will produce the equivalent of "SELECT count(*) FROM EMP".

VIEW support for aggregate functions

The SQL view engine supports PROP:GroupBy and PROP:Having. These properties allow you to add respectively GROUP BY and HAVING SQL clauses to your SELECT statement.

Note:

PROP:GroupBy must be set first to allow PROP:Having to be generated.

Example:

```

PROGRAM

MAP
END

EMP      FILE,DRIVER('ORACLE'),NAME('EMP'),PRE(EMP)
P_EKY_EMP  KEY(EMP:EMPNO),NOCASE,OPT,PRIMARY
KEY_DEP    KEY(EMP:DEPTNO),DUP,NOCASE,OPT
Record    RECORD
EMPNO      SHORT          !Emp-no
ENAME      CSTRING(11)    !Employee name
JOB        CSTRING(10)    !Job
HIREDATE   DATE          !Hiredate
MGR        SHORT          !Manager
SAL        PDECIMAL(7,2) !Salary
COMM       PDECIMAL(7,2) !Commisison
DEPTNO     BYTE
          END
          END

MyView VIEW(EMP)
        PROJECT(EMP:Mgr)
        PROJECT(EMP:Sal)
        END

CODE
OPEN(EMP)
OPEN(MyView)
MyView{'EMP:Sal',PROP:Name} = 'sum(sal)'
MyView{PROP:GroupBy} = 'Mgr'
MyView{PROP:Having} = 'sum(sal) > 100000'

SET(MyView)
NEXT(MyView)

```

The previous example code is the equivalent to "SELECT mgr, sum(sal) FROM EMP GROUP BY mgr HAVING sum(sal) > 100000"

In other words, this code will return a list of all Manager IDs and the total salary of their subordinates if their subordinates make a total of more than 100000.

Debugging Your SQL Application

All of the SQL Accelerator drivers can create a log file documenting Clarion I/O statements they process, the corresponding SQL statements, and the SQL return codes.

You can generate system-wide logs and on-demand logs (conditional logging based on your program logic).

System-wide Logging

A new utility/example application has been added--Trace.EXE. You can run this from the Clarion Start Menu option. A compiled version is installed in the .BIN directory and the source .APP is installed in the \Examples\Resource\Trace directory. This utility allows you to easily set tracing options for each file driver and for the VIEW engine.

These settings are stored in WIN.INI.

For system-wide logging, you can add the following to your WIN.INI file:

```
[CWdriver]
Profile=[1|0]
Details=[1|0]
Trace=[1|0]
TraceFile=[Pathname]
```

where *driver* is the database driver name (for example [CWTopSpeed]). Neither the INI section name [CWdriver] nor the INI entry names are case sensitive.

Profile=1 tells the driver to include the Clarion I/O statements in the log file; Profile=0 tells the driver to omit Clarion I/O statements. The Profile switch must be turned on for the Details switch to have any effect.

Details=1 tells the driver to include record buffer contents in the log file; however, if the file is encrypted, you must turn on both the Details switch and the ALLOWDETAILS switch to log record buffer contents (see *ALLOWDETAILS*). Details=0 tells the driver to omit record buffer contents. The Profile switch must be turned on for the Details switch to have any effect.

Trace=1 tells the driver to include all calls to the back-end file system, including the generated SQL statements and their return codes, in the log file. Trace=0 omits these calls. The Trace switch generally generates log information that helps to debug the SQL drivers, but is normally not particularly useful to the developer.

TraceFile names the log file to write to. If TraceFile is omitted the driver writes the log to *driver.log* in the current directory. *Pathname* is the full pathname or the filename of the log file to write. If no path is specified, the driver writes the specified file to the current directory.

Logging opens the named logfile for exclusive access. If the file exists, the new log data is appended to the file.

On Demand Logging

For on-demand logging you can use property syntax within your program to conditionally turn various levels of logging on and off. The logging is effective for the target table and any view for which the target table is the primary table.

```
file{PROP:Profile}=Pathname    !Turns Clarion I/O logging on
file{PROP:Profile}=''         !Turns Clarion I/O logging off
PathName = file{PROP:Profile} !Queries the name of the log file
file{PROP:Log}=string         !Writes the string to the log file
file{PROP:Details}=1         !Turns Record Buffer logging on
fFile{PROP:Details}=0        !Turns Record Buffer logging off
```

where *Pathname* is the full pathname or the filename of the log file to create. If you do not specify a path, the driver writes the log file to the current directory.

You can also accomplish on demand logging with a SEND() command and the LOGFILE driver string. See *LOGFILE* for more information.

Language Level Error Checking

You can use the FILEERROR() and FILEERRORCODE() functions to capture messages and codes returned from the backend server to the SQL Accelerator driver. See the *Language Reference* for more information on these functions.

SQL Accelerator Drivers:Supported Commands and Attributes

File Attributes	Supported
CREATE	Y
DRIVER(<i>filetype</i> [, <i>driver string</i>])	Y
NAME	Y
ENCRYPT	N
OWNER(<i>password</i>)	Y ₁
RECLAIM	N
PRE(<i>prefix</i>)	Y
BINDABLE	Y
THREAD	Y
EXTERNAL(<i>member</i>)	Y
DLL(<i>flag</i>)	Y
OEM	N
File Structures	Supported
INDEX	Y
KEY	Y
MEMO	N
BLOB	Y
RECORD	Y
Index, Key, Memo Attributes Supported	
BINARY	N ₃
DUP	Y
NOCASE	Y
OPT	N

PRIMARY	Y
NAME	Y
Ascending Components	Y
Descending Components	Y
Mixed Components	Y
Field Attributes	Supported
DIM	N
OVER	Y
NAME	Y
File Procedures	Supported
BOF(<i>file</i>)	N
BUFFER(<i>file</i>)	Y
BUILD(<i>file</i>)	Y
BUILD(<i>key</i>)	Y
BUILD(<i>index</i>)	Y ₃
BUILD(<i>index, components</i>)	Y ₃
BUILD(<i>index, components, filter</i>)	N
BYTES(<i>file</i>)	Y
CLOSE(<i>file</i>)	Y
COPY(<i>file, new file</i>)	N
CREATE(<i>file</i>)	Y
DUPLICATE(<i>file</i>)	Y
DUPLICATE(<i>key</i>)	Y
EMPTY(<i>file</i>)	Y
EOF(<i>file</i>)	N
FLUSH(<i>file</i>)	N
LOCK(<i>file</i>)	N

NAME(<i>label</i>)	Y
OPEN(<i>file, access mode</i>)	Y
PACK(<i>file</i>)	N
POINTER(<i>file</i>)	N
POINTER(<i>key</i>)	N
POSITION(<i>file</i>)	N
POSITION(<i>key</i>)	Y
RECORDS(<i>file</i>)	Y
RECORDS(<i>key</i>)	Y
REMOVE(<i>file</i>)	Y
RENAME(<i>file, new file</i>)	N
SEND(<i>file, message</i>)	Y
SHARE(<i>file, access mode</i>)	Y
STATUS(<i>file</i>)	Y
STREAM(<i>file</i>)	N
UNLOCK(<i>file</i>)	N
Record Access	Supported
ADD(<i>file</i>)	Y
ADD(<i>file, length</i>)	N
APPEND(<i>file</i>)	Y
APPEND(<i>file, length</i>)	N
DELETE(<i>file</i>)	Y
GET(<i>file, key</i>)	Y
GET(<i>file, filepointer</i>)	N
GET(<i>file, filepointer, length</i>)	N
GET(<i>key, keypointer</i>)	N
HOLD(<i>file</i>)	N

NEXT(<i>file</i>)	Y
NOMEMO(<i>file</i>)	N
PREVIOUS(<i>file</i>)	Y
PUT(<i>file</i>)	Y
PUT(<i>file</i> , <i>filepointer</i>)	N
PUT(<i>file</i> , <i>filepointer</i> , <i>length</i>)	N
RELEASE(<i>file</i>)	N
REGET(<i>file</i> , <i>string</i>)	N
REGET(<i>key</i> , <i>string</i>)	Y
RESET(<i>file</i> , <i>string</i>)	N
RESET(<i>key</i> , <i>string</i>)	Y
SET(<i>file</i>)	Y
SET(<i>file</i> , <i>key</i>)	N
SET(<i>file</i> , <i>filepointer</i>)	N
SET(<i>key</i>)	Y
SET(<i>key</i> , <i>key</i>)	Y
SET(<i>key</i> , <i>keypointer</i>)	N
SET(<i>key</i> , <i>key</i> , <i>filepointer</i>)	N
SKIP(<i>file</i> , <i>count</i>)	Y
WATCH(<i>file</i>)	Y
Transaction Processing	Supported (see Note 2)
LOGOUT(<i>timeout</i> , <i>file</i> , ..., <i>file</i>)	Y ⁴
COMMIT	Y
ROLLBACK	Y
Null Data Processing	Supported
NULL(<i>field</i>)	Y
SETNULL(<i>field</i>)	Y

SETNONNULL(*field*) Y

Notes

- 1 We recommend using a variable password that is lengthy and contains special characters because this more effectively hides the password value from anyone looking for it. For example, a password like "dd...#\$...*&" is much more difficult to "find" than a password like "SALARY."

Tip

To specify a variable instead of the actual password in the Owner Name field of the File Properties dialog, type an exclamation point (!) followed by the variable name. For example: !MyPassword.

- 2 See also *PROP:Logout* in the *Language Reference*.
- 3 BUILD(index) sets internal driver flags to guarantee the driver generates the correct ORDER BY clause. The driver does not call the backend server.
- 4 Whether LOGOUT also LOCKs the table depends on the server's configuration for transaction processing. See your server documentation.

CHECKFORNULL

The CHECKFORNULL field switch applies to all SQL drivers

Usage:

In the External name attribute:

'field name | CHECKFORNULL'

When browsing through a table, it is sometimes necessary for the driver to request all rows that are at, or before, the current row. It does this by generating a WHERE clause. For example:

```
WHERE (field1 <= value) AND (field1 < value OR field2 <= value2)
```

The above example is for a two component key. For more components, the WHERE clause gets longer, and this will work well in most cases. However, in SQL, if a field has a NULL value, then field < value is false, field = value is false, and field > value is also false. So, if you are sorting on field components that contain NULL values, you need to set the external field name of the field to

'field name | CHECKFORNULL'

This will force the driver to generate:

```
WHERE((field1<=value OR field1 IS NULL))AND((field1<value OR field1 IS NULL)OR field2<=value2)
```

So, in this example, the WHERE clause will also return rows that contain NULL values, instead of rejecting them.

SQL Driver Strings(Generic)

SQL Driver Strings

There are switches or "driver strings" you can set to control the way your application creates, reads, and writes files with a specific driver. Driver strings are simply messages or parameters that are sent to the file driver at run-time to control its behavior. See *Common Driver Features--Driver Strings* for an overview of these runtime Database Driver switches and parameters.

Tip

A forward slash precedes all SQL driver strings. The slash allows the driver to distinguish between driver strings and SQL statements sent with the SEND function.

The SQL Accelerator Drivers support the following Driver Strings:

ALLOWDETAILS
APPENDBUFFER
BINDCOLORDER
BINDCONSTANTS
CLIPSTRINGS
FASTCOLUMNFETCH
FORCEUPPERCASE
GATHERATOPEN
GETINFO
IGNORETRUNCATION
ISOLATIONLEVEL
JOINTYPE
LOGFILE
NESTING
ODBCCALL
ORDERINSELECT
USEINNERJOIN
VERIFYVIASELECT
WHERE
ZERODATE
ZEROISNULL

ALLOWDETAILS

DRIVER('SQLDriver', '/ALLOWDETAILS = TRUE | FALSE')

The ALLOWDETAILS driver string allows the SQL Accelerator driver to include record buffer contents in the log file for encrypted files.

The ALLOWDETAILS driver string works with the Details switch described in the *Debugging Your SQL Application* section.

APPENDBUFFER

DRIVER('SQLDriver', '/APPENDBUFFER = size ')

[Buffer" =] SEND(file, 'APPENDBUFFER [= size]')

By default, APPEND adds records to the file one at a time. To get better performance over a network you can tell the driver to build up a buffer of records then send all of them to Btrieve at once. Size is the number of records you want to allocate for the buffer. SEND returns the number of records that will fit in the buffer.

BINDCOLORDER

DRIVER('SQLDriver', '/BINDCOLORDER = [0 | 1 | 2]')

Valid for all drivers *except* Oracle

When executing a SELECT statement the driver has to do the following:

1. Compile the SELECT statement
2. Bind memory locations for the columns to be returned
3. Bind memory locations for the WHERE clause
4. Executes the SELECT statement
5. Fetch the data

The order that these are executed is not completely fixed. The compile (Step 1) must be done first and the fetch (Step 5) last. However, the other three steps can be executed in any order.

If BINDCOLORDER is set to 0, the order is 1, 2, 3, 4, 5.

If BINDCOLORDER is set to 1, the order is 1, 3, 2, 4, 5.

If BINDCOLORDER is set to 2, the order is 1, 3, 4, 2, 5.

The default is 0 for all supported driver *except* MySQL, which has a default setting of 2.

BINDCONSTANTS

```
DRIVER('SQLDriver', '/BINDCONSTANTS = TRUE | FALSE ')
```

```
[ Bind" = ] SEND(file, '/BINDCONSTANTS [ = TRUE | FALSE ]')
```

(NOTE: Not valid for ORACLE Accelerator)

By default (BINDCONSTANTS=TRUE) the SQL Accelerator binds memory locations instead of generating text equivalents for constant values. However, some back ends get confused when doing this. So if you find that your SQL based BrowseBox will not scroll, or your backend returns incorrect results for a BrowseBox you can turn off binding of constant values by setting BINDCONSTANTS to FALSE.

CLIPSTRINGS

```
DRIVER('SQLDriver', '/CLIPSTRINGS = TRUE | FALSE ')
```

```
[ Clipped" = ] SEND(file, '/CLIPSTRINGS [ = TRUE | FALSE ]')
```

(NOTE: Not valid for ORACLE Accelerator)

By default (CLIPSTRINGS=TRUE), the SQL driver CLIPs strings before sending them to the backend server (see *CLIP* in the *Language Reference*). To send the full (unclipped) string, set CLIPSTRINGS=FALSE.

FASTCOLUMNFETCH

```
DRIVER('SQLDriver', '/FASTCOLUMNFETCH = TRUE | FALSE ')
```

```
[ Fetch" = ] SEND(file, '/FASTCOLUMNFETCH [ = TRUE | FALSE ]')
```

(NOTE: Not valid for ORACLE Accelerator)

By default, the SQL driver will attempt to use the extended fetch abilities of a back end to retrieve column information. Some back ends support extended fetch, but not when fetching column information. To stop these back ends from crashing or returning invalid errors when opening a file, you can set */FASTCOLUMNFETCH=FALSE*.

FORCEUPPERCASE

DRIVER('SQLDriver', '/FORCEUPPERCASE = TRUE | FALSE ')

[Uppered" =] SEND(file, '/FORCEUPPERCASE [= TRUE | FALSE]')

(NOTE: Not valid for ORACLE Accelerator)

By default (FORCEUPPERCASE=FALSE), the SQL Driver passes the table name in mixed case to the SQLColumns function to verify the existence of the table. However, some backends require the table name to be passed in uppercase. To pass the table name in uppercase, set FORCEUPPERCASE=TRUE. See also *VERIFYVIASELECT*.

GATHERATOPEN

DRIVER('SQLDriver', '/GATHERATOPEN = TRUE | FALSE ')

(NOTE: Not valid for ORACLE Accelerator)

By default the driver delays gathering field information until it is required. However, some backends (like Sybase 11) perform poorly under these conditions. Setting GATHERATOPEN to TRUE forces the driver to gather most of the field information when the file is opened, which avoids a slowdown during program execution.

GETINFO

Result = filelabel{PROP:GETINFO, property}

(NOTE: Not valid for ORACLE Accelerator)

Use this property to retrieve information about a connection to any ODBC data source. The full list of available ODBC properties can be found in ODBCATTR.INC

IGNORETRUNCATION

DRIVER('SQLDriver', '/IGNORETRUNCATION = TRUE | FALSE ')

filelabel{PROP:IGNORETRUNCATION} = TRUE | FALSE

(NOTE: Not valid for ORACLE Accelerator)

You can declare your string data to be a different size in Clarion than on the server. For example, you can define a CHAR(10000) as CSTRING(50) if all you are interested in is the first 50 characters of the data. However, doing this will generate an ODBC warning about string truncation.

By default, Clarion treats this as a normal error. If you want to ignore this warning, you can set IGNORETRUNCATION=TRUE, or alternatively, use filelabel{PROP:IgnoreTruncation} = 1.

ISOLATIONLEVEL

```
DRIVER('SQLDriver', '/ISOLATIONLEVEL = number' )
number = SEND(file, '/ISOLATIONLEVEL = number')
file{PROP:IsolationLevel} = number
number = file{PROP:IsolationLevel}
```

(NOTE: Not valid for ORACLE Accelerator)

The following terms are used to define transaction isolation levels:

<i>Dirty Read</i>	Transaction 1 changes a row. Transaction 2 reads the changed row before transaction 1 commits the change. If transaction 1 rolls back the change, transaction 2 will have read a row that is considered to have never existed.
<i>Nonrepeatable Read</i>	Transaction 1 reads a row. Transaction 2 updates or deletes that row and commits this change. If transaction 1 attempts to reread the row, it will receive different row values or discover that the row has been deleted.
<i>Phantom</i>	Transaction 1 reads a set of rows that satisfy some search criteria. Transaction 2 inserts a row that matches the search criteria. If transaction 1 reexecutes the statement that read the rows, it receives a different set of rows.

number must be one of the following values:

- 1 Dirty reads, non-repeatable reads, and phantoms are possible.
- 2 Dirty reads are not possible. Non-repeatable reads and phantoms are possible.
- 4 Dirty reads and non-repeatable reads are not possible. Phantoms are possible.
- 8 Transactions are serializable. Dirty reads, non-repeatable reads, and phantoms are not possible.
- 16 Transactions are serializable, but higher concurrency is possible than with 8. Dirty reads are not possible.

Typically, 8 is implemented by using locking protocols that reduce concurrency and 16 is implemented by using a non-locking protocol such as record versioning. Oracle's Read Consistency isolation level is an example of 16.

By default, the SQL driver set the transaction isolation level to what is set in the data source. The other drivers set it to 1.

The return number is the current value of the isolation level. A zero return indicates the file is not connected to a database.

IsolationLevel 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

JOINTYPE

```
DRIVER('SQLDriver', '/JOINTYPE = Watcom | DB2 | Microsoft | FirstSQL | Inner | None' )
```

```
[ Join" = ] SEND(file, '/JOINTYPE [ = Watcom | DB2 | Microsoft | FirstSQL | Inner | None ]' )
```

(NOTE: Not valid for ORACLE Accelerator)

The SQL standard does not support joins to more than one child (or parent). Most vendors consider this limitation unacceptable and have extended the standard. However, they have done so in different ways. The SQL driver attempts to determine the join type used by the backend, but if it does not get it right, then you should use the JOINTYPE driver string in the primary file of the view. Note that specifying *Inner* is normally slower than *Watcom*, *DB2*, *Microsoft* or *FirstSQL* and *None* is slower than *Inner*, but will work with all backends because the join is done on the client.

When using ODBC, the ODBC 3.0 standard does support multiple joins, so ODBC 3.0 compliant drivers should not require this switch.

JOINTYPE=DB2

This is the join syntax used by IBM's DB2. This generates ANSI compliant outer joins. The Base Normal Form for the relevant DB2 specification is:

```
from-clause ::=
    FROM <table-ref>

table-ref ::=
    <single-table> |
    <joined-table>

single-table ::=
    table-name AS correlation-name

joined-table ::=
    <table-ref> LEFT OUTER JOIN <single-table> ON join-
condition
```

JOINTYPE=MICROSOFT

This is the join syntax specified by the ODBC 2.0 spec. The Base Normal Form for the relevant ODBC spec is:

```

from-clause ::=
    FROM <table-ref> |
    FROM <odbc-joined-table>

table-ref ::=
    <single-table> |
    <joined-table>

single-table ::=
    table-name AS correlation-name

odbc-joined-table ::=
    {oj <joined-table> }

joined-table ::=
    <single-table> LEFT OUTER JOIN <table-ref> ON join-
condition

```

JOINTYPE=WATCOM

This is the join syntax used by SQL Anywhere and is a merger of the ODBC and ANSI specifications. The Base Normal Form for this syntax is:

```

from-clause ::=
    FROM <table-ref> |
    FROM <odbc-joined-table>

table-ref ::=
    <single-table> |
    <joined-table>

single-table ::=
    table-name AS correlation-name

odbc-joined-table ::=
    {oj <joined-table> }

joined-table ::=
    <table-ref> LEFT OUTER JOIN <single-table> ON join-
condition

```

JOINTYPE=FIRSTSQL

This is the join syntax used by FirstSQL and is not recommended to be used with any other file format.

JOINTYPE=INNER

This is a format that should work with any database, but is likely to be very slow.

JOINTYPE=NONE

This indicates to perform the join on the client.

LOGFILE

DRIVER('SQLDriver', '/LOGFILE [= Pathname] [[message]]')

[LogFile" =] SEND(file, '/LOGFILE [= Pathname] [[message]]')

The LOGFILE driver string turns logging on and off, and optionally writes a message to the log file. Turning the LOGFILE switch on writes Clarion I/O statements processed by the driver to the specified log file. The LOGFILE driver string is equivalent to the Profile switch described in the *Debugging Your SQL Application* section.

Pathname is the full pathname or the filename of the log file to write. If you do not specify a path, the driver writes the log file to the current directory. If Pathname is omitted, the driver writes the log to SQLDriver.log in the current directory.

If the log file already exists, the driver appends to it; otherwise, the driver creates the log file.

The *message* is optional, however, if included, it must be surrounded by square brackets ([]) and a space must precede the opening square bracket.

Note: /LOGFILE must be the last driver string specified by the DRIVER attribute.

NESTING

```
DRIVER('SQLDriver', '/NESTING = TRUE | FALSE ' )
```

```
[ Nest" = ] SEND(file, '/NESTING [ = TRUE | FALSE ]' )
```

(NOTE: Not valid for ORACLE Accelerator)

Some SQL drivers do not support parent->child->grandchild style views. The SQL driver attempts to determine if this is supported. If the driver does not get it right and the backend does not support these type of views, then you need to set NESTING=FALSE. This causes the join to be done on the client.

ODBCCALL

```
DRIVER('SQL Driver', '/ODBCCALL = TRUE | FALSE ' )
```

```
[ Call" = ] SEND(file, '/ODBCCALL [ = TRUE | FALSE ]' )
```

By default (ODBCCALL = True) the SQL Accelerator reformats your CALL statements to match the ODBC standard call syntax. To disable this automatic reformatting, set ODBCCALL=FALSE.

ORDERINSELECT

```
DRIVER('SQLDriver', '/ORDERINSELECT= TRUE | FALSE' )
```

```
[ OIS" = ] SEND(file, '/ORDERINSELECT [ = TRUE | FALSE ]' )
```

(NOTE: Not valid with the Oracle Accelerator)

Some backends require that any fields used in the ORDER BY clause also appear in the SELECT statement. By setting this property to true the driver will make sure this rule is applied for all views regardless of the fields projected. You can also read PROP:OrderInSelect to get its current value

USEINNERJOIN

```
DRIVER('SQLDriver', '/USEINNERJOIN= TRUE | FALSE' )
```

```
[ Join" = ] SEND(file, '/USEINNERJOIN [ = TRUE | FALSE ]' )
```

By default (USEINNERJOIN = True) the SQL Accelerator generates the following ANSI SQL for inner joins:

```
SELECT ... FROM table1 INNER JOIN table2 ON table1.field=table2.field
```

However, not all backends support ANSI SQL. The driver provides an alternative syntax for inner joins. To generate the following alternative syntax, set USEINNERJOIN=FALSE:

```
SELECT ... FROM table1, table2 WHERE table1.field=table2.field
```

VERIFYVIASELECT

DRIVER('SQL Driver', '/VERIFYVIASELECT = TRUE | FALSE')

[Verify" =] SEND(file, '/VERIFYVIASELECT [= TRUE | FALSE]')

VERIFYVIASELECT lets the SQL Driver use an alternative, sometimes faster, method to validate fields when opening a table. By default (VERIFYVIASELECT=FALSE), the SQL Driver uses the SQLColumns function to validate fields. However, some backends (particularly SQL Anywhere) can validate fields faster using a SELECT statement. To verify fields using the SELECT statement, set VERIFYVIASELECT to TRUE.

VERIFYVIASELECT defaults to TRUE for SQL Anywhere backends.

WHERE (SQL Driver String)

```
[ Where" = ] SEND (file, '/WHERE [ where-clause ]')
```

The SQL Accelerator drivers automatically build SQL WHERE clauses when your Clarion code contains a SET followed by a NEXT or PREVIOUS. You can customize the driver generated WHERE clause by using the WHERE driver string. You can also set the WHERE driver string at runtime with the use of PROP:WHERE. PROP:WHERE is a write-only property.

The SEND must be executed after the SET statement and before the NEXT or PREVIOUS statement.

Note: The SET statement clears any WHERE clause set by the SEND statement.

Because the SQL driver's generated SELECT statement is not compiled until the NEXT or PREVIOUS statement, the SEND function posts no error code and returns no result. For example:

```
Orders    FILE,PRE(Ord),DRIVER('ODBC'),NAME('Ord')
NameDate  KEY(+Ord:NameId,-Ord:Date)
Record    RECORD
Name      STRING(12),NAME('NameId')
Date      DATE,NAME('OrderDate')
Type      STRING(1),NAME('OrderType')
Details   STRING(20),NAME('OrderDetails')
          END
          END

CODE
Ord:Name = 'SMITH'
SET(Ord:NameDate,Ord:NameDate)
SEND(Ord, '/WHERE OrderType = "M"')
!or you can use Ord{PROP:WHERE} = Ord{PROP:WHERE} & 'AND OrderType = "M"'
LOOP
    NEXT(Orders)
    !...some processing
END
```

This generates a SELECT statement similar to:

```
SELECT NameId,OrderDate,OrderType,OrderDetails FROM Orders
WHERE (NameID >= 'SMITH') AND (OrderType = 'M')
```

ZERODATE

DRIVER("SQLDriver", '/ZERODATE = NULL | TRUE | FALSE')

[Nulls" =] SEND(file, '/ZERODATE [= NULL | TRUE | FALSE]')

ZERODATE defines how the target driver should generate a WHERE clause for cleared DATE and TIME fields and replaces the \ZEROISNULL driver string.

ZERODATE=NULL is equivalent to ZEROISNULL=TRUE and is the default behavior.

ZERODATE=0 (FALSE) is equivalent to ZEROISNULL=FALSE.

ZERODATE=1 (TRUE) indicates that a cleared date will be generated as 01/01/0001 and a cleared time is generated as 0.

If both ZERODATE and ZEROISNULL is specified in the driver string, the last one will be used.

If you use the driver string editor in the dictionary editor, it will automatically convert ZEROISNULL to the equivalent ZERODATE.

ZEROISNULL

DRIVER("SQLDriver", '/ZEROISNULL = TRUE | FALSE')

[Nulls" =] SEND(file, '/ZEROISNULL [= TRUE | FALSE]')

(NOTE: Not valid for ORACLE Accelerator)

ZEROISNULL lets the SQL Accelerator Drivers set DATE and TIME fields to zero (0) rather than null. By default (ZEROISNULL=TRUE), the SQL Accelerator Drivers assumes a DATE or TIME field with a value of zero (0) should be a null value in the backend database, and adjusts the values to NULL when writing to the backend. To allow the driver to set DATE and TIME fields to zero rather than null, set ZEROISNULL to FALSE.

SQL Driver Properties(Generic)

PROP:Alias

PROP:Alias sets or returns the alias the SQL Accelerator driver uses when generating SELECT statements for a view. PROP:Alias only returns a value previously set using PROP:Alias. For example:

```
Customer{PROP:Alias} = 'C'           !set new table alias
OldAlias" = Customer{PROP:Alias} = '' !use default alias
```

Tip: Prior to an assignment to PROP:Alias, the return value for PROP:Alias is an empty string.

The SQL driver generates an SQL statement which uses an Alias of "A" for the first file in the View, "B" for the second etc. If you wish to use Prop:SQLFilter your filter has to be compatible with the previously generated SQL statement - ergo: you need use A/B/etc as the file prefixes. You can use file{PROP:Alias} to specify what alias the driver should use when constructing SELECT statements for views.

PROP:AlwaysRebind

PROP:AlwaysRebind sets or returns the toggle that controls whether the SQL Accelerator rebinds memory locations when a NULL state changes.

For all backends except MSSQL, PROP:AlwaysRebind defaults to 0 or False, so the SQL driver does not rebind memory locations when a NULL state changes. However, some SQL backends (including MSSQL) do not recheck the null state, so they must have the memory location rebound to force the change of null state. Setting PROP:AlwaysRebind to 1 or True tells the SQL Accelerator to do this extra binding.

PROP:ConnectionString

PROP:ConnectionString returns an SQL database's connection information. For example:

```
AFileOwner STRING(256)
AFile FILE,DRIVER('ODBC'),OWNER(AFileOwner)
CODE
AFileOwner='DataSource'
OPEN(Afile)
IF NOT ERRORCODE()
  AFileOwner=AFile{PROP:ConnectionString}
END
```

PROP:DBMSver

Good for all SQL Drivers except Oracle.

File{PROP:DBMSver} returns a character string indicating the version of the DBMS accessed by the driver. The version is of the form `##.##.####`, where the first two digits are the major version, the next two digits are the minor version, and the last four digits are the release version.

PROP:Details

See the Details switch described in the *Debugging Your SQL Application* section.

PROP:Disconnect

PROP:Disconnect CLOSEs any open files in the target file's database, then disconnects the application from the database.

EXAMPLE:

```
FileLabel{PROP:Disconnect} !No equal sign needed
```

PROP:GroupBy, PROP:Having

The SQL view engine supports PROP:GroupBy and PROP:Having. These properties allow you to add respectively GROUP BY and HAVING clauses to your SELECT statement. Note that PROP:GroupBy must be set first to allow PROP:Having to be generated.

See Also:

```

PROGRAM

MAP
END

EMP      FILE,DRIVER('ORACLE'),NAME('EMP'),PRE(EMP)
P_EKY_EMP  KEY(EMP:EMPNO),NOCASE,OPT,PRIMARY
KEY_DEP   KEY(EMP:DEPTNO),DUP,NOCASE,OPT
Record    RECORD
EMPNO     SHORT           !Emp-no
ENAME     CSTRING(11)     !Employee name
JOB       CSTRING(10)    !Job
HIREDATE  DATE           !Hiredate
MGR       SHORT           !Manager
SAL       PDECIMAL(7,2)  !Salary
COMM      PDECIMAL(7,2)  !Commisison
DEPTNO    BYTE
          END
          END

MyView VIEW(EMP)
  PROJECT(EMP:MGR)
  PROJECT(EMP:SAL)
END

CODE
  OPEN(EMP)
  OPEN(MyView)
  MyView{'EMP:SAL',PROP:Name} = 'sum(sal)'
  MyView{PROP:GroupBy} = 'MGR'
  MyView{PROP:Having} = 'sum(sal) > 100000'
SET(MyView)
  NEXT(MyView)

```

The example code above is the equivalent to "SELECT mgr, sum(sal) FROM EMP GROUP BY mgr HAVING sum(sal) > 100000"

In other words, this code will return a list of all Manager IDs and the total salary of their subordinates if their subordinates make a total of more than 100000.

PROP:hdbc

(NOTE: Not valid for ORACLE Accelerator)

PROP:hdbc returns the current hdbc used by the SQL driver. Thus ?MyFile{PROP:hdbc} may be used for ODBC API calls requiring hdbc.

PROP:henv

(NOTE: Not valid for ORACLE Accelerator)

PROP:henv returns the current henv used by the SQL driver. Thus ?MyFile{PROP:henv} may be used for ODBC API calls requiring henv. For example, the SQLDescribeCol function:

```
rc# = SQLDataSources(Myfile{PROP:henv},SQL_FETCH_NEXT,ODBC:driver, |
                    drvLen,drvlen,ODBC:Description,desclen,desclen)
```

PROP:Hint

The HINT driver string is valid for Oracle and MS-SQL drivers..

Oracle support for PROP:HINT

MS-SQL support for PROP:HINT

PROP:hstmt

(NOTE: Not valid for ORACLE Accelerator)

PROP:hstmt returns the current hstmt used by the SQL driver. Thus ?MyFile{PROP:hstmt} may be used for ODBC API calls requiring hstmt. For example, the SQLDescribeCol function:

```
Myfile{PROP:SQL} = 'Select * from ATable'
rc# = SQLDescribeCol(Myfile{PROP:hstmt},Num,Name,Max,NameL, |
                    Type,Def,Scale,Null)
```

PROP:Inner

PROP:Inner is a writable property for SQL Accelerator drivers. This is useful for testing the ODBC USEINNERJOIN driver string. See *PROP:Inner* in the *Language Reference* for more information.

PROP:IsolationLevel

PROP:IsolationLevel is used to define transaction isolation levels. See ISOLATIONLEVEL

PROP:LogonScreen

PROP:LogonScreen sets or returns the value that determines whether the driver automatically prompts for logon information. By default, PROP:LogonScreen=TRUE, and the driver displays a logon window if no connect string is supplied. If set to FALSE, and there is no connect string, the OPEN(file) fails and FILEERRORCODE() returns a driver specific error code. For example:

```
Afile FILE,DRIVER('SQLAnywhere')
!file declaration with no userid and password
    END

CODE
    Afile{PROP:LogonScreen}=True                !enable auto login screen
    OPEN(Afile)
```

In the above example, the logon screen uses the SQLAnywhere Connect dialog. Consult your specific database documentation for more information on this dialog. The end-user's ability to use the connect dialog will depend on the security surrounding the specific database. For example, the end-users may have access rights to a named database that they can access with the database's client software, but they may not have access rights to other files that comprise the database.

Note:

PROP:LogonScreen is valid for all SQL based drivers.

PROP:Log

PROP:Log writes a string to the log file. For example:

```
Afile FILE,DRIVER('ODBC'),OWNER('DataSource')
CODE
    OPEN(Afile)
    IF NOT ERRORCODE()
        Afile{PROP:Log}='Afile opened:&CLOCK()
    END
```

PROP:LogFile

Same as PROP:Profile -- or backward compatibility.

PROP:LoginTimeout

(NOTE: Not valid for ORACLE Accelerator)

PROP:LoginTimeout sets a time limit in seconds for an SQL database's login screen. If the user does not respond in the allotted time, the connection fails and the login is aborted. The default is to wait indefinitely for user input. Some servers do not support this feature and may ignore the instruction. For example:

```
AFile FILE,DRIVER('SQL Driver'),OWNER('DataSource')
CODE
OPEN(Afile)
IF NOT ERRORCODE()
  AFile{PROP:LoginTimeOut}=60 !allow 1 minute for login
END
```

PROP:OrderAllTables

Setting PROP:OrderAllTables to True forces the SQL Accelerator driver to use linking fields and secondary files' key component fields, as well as the primary file's key component fields, in the ORDER BY clause it sends to the server. You may need this switch if you are using a Clarion VIEW that joins multiple tables. By default (View{PROP:OrderAllTables}=FALSE), the SQL Accelerator driver includes only the primary file's key components in the ORDER BY clause it sends to the SQL server. For example:

```
BRW1::View:Browse VIEW(Customer)
    PROJECT(CUST:CustNo)
    PROJECT(CUST:Name)
    PROJECT(CUST:Zip)
    PROJECT(CUST:CustNo)
    JOIN(ORD:ByCustomer,CUST:CustNo)
    PROJECT(ORD:OrderNo)
    PROJECT(ORD:OrderDate)
    END
END
CODE
  ?BRW1::View:Browse{PROP:OrderAllTables} = TRUE
```

Accessing this VIEW then generates a SELECT statement similar to:

```
SELECT CustNo,Name,Zip,OrderNo,OrderDate FROM Customer,Ord
  WHERE (Customer.CustNo = Ord.CustNo)
  ORDER BY CustNo,OrderNo
```

PROP:OrderInSelect

(NOTE: Not valid for ORACLE Accelerator)

Some SQL backends require that any fields used in the ORDER BY clause also appear in the SELECT statement. By setting this property to TRUE (1) the driver will make sure that this rule is applied for all views regardless of the fields that you project.

PROP:Profile

Setting PROP:Profile to true tells the driver to include Clarion I/O statements in the log file. See the Profile switch described in the *Debugging Your SQL Application* section.

Profile=1 tells the driver to include the Clarion I/O statements in the log file; Profile=0 tells the driver to omit Clarion I/O statements. The Profile switch must be turned on for the Details switch to have any effect.

Details=1 tells the driver to include record buffer contents in the log file; however, if the file is encrypted, you must turn on both the Details switch and the /ALLOWDETAILS switch to log record buffer contents (see *ALLOWDETAILS*). Details=0 tells the driver to omit record buffer contents. The Profile switch must be turned on for the Details switch to have any effect.

Note: /ALLOWDETAILS is only valid as a parameter of the DRIVER attribute (Driver Options field in the File Properties dialog). It is not valid with the SEND command.

PROP:QuoteString

(NOTE: Not valid with Oracle Accelerator.)

PROP:QuoteString sets or returns the column name delimiter (typically a quote) that the SQL Accelerator Driver uses to surround column names within its generated SQL statements. Different backends require different delimiter characters.

You can use PROP:QuoteString to build your own dynamic SQL statements. Note that you must enclose any column names that are also SQL reserved words in the correct delimiter character. See *Using Embedded SQL*.

Some backends do not correctly return the delimiter character. For those backends you should set the value of PROP:QuoteString before using it.

PROP:SQL

You can use Clarion's property syntax (PROP:SQL) to execute SQL statements in your program code by using PROP:SQL and naming the FILE or imported SQL VIEW in the data dictionary as the *target* within the normal execution of your program. This is only valid when using an SQL file driver (such as the ODBC, Scalable SQL, or Oracle drivers). You can send any SQL statements supported by the SQL server.

This capability lets your program do backend operations independent of the SQL Accelerator driver's generated SQL. For example, multi-record updates can often be accomplished more efficiently with a single SQL statement than with a template generated Process procedure that updates one record at a time. In cases like these it makes sense for you to take control and send custom SQL statements to the backend, and PROP:SQL lets you do this.

If you issue an SQL statement that returns a result set (such as an SQL SELECT statement), you use NEXT(file) to retrieve the result set one row at a time, into the file's record buffer. The FILE declaration receiving the result set must have the same number of fields as the SQL SELECT statement will return. If the Clarion ERRORCODE procedure returns 90, the FILEERRORCODE() and FILEERROR() functions return any error code and error message set by the back-end SQL server.

You may also query the contents of PROP:SQL to get the last SQL statement issued by the file driver.

Examples:

```
SQLFile{PROP:SQL}='SELECT field1,field2 FROM table1'      |
                  & 'WHERE field1 > (SELECT max(field1)' |
                  & 'FROM table2'          ! Returns a result set you
                                          ! get one row at a time
                                          ! using NEXT(SQLFile)

!Call stored procedure
SQLFile{PROP:SQL}='CALL GetRowsBetween(2,8)''

!No result set
SQLFile{PROP:SQL}='CREATE INDEX ON table1(field1 DESC)''

!DBA tasks
SQLFile{PROP:SQL}='GRANT SELECT ON mytable TO fred''

!Get last SQL statement
SQLString=SQLFile{PROP:SQL}
```

SEND

You can use the Clarion SEND procedure to send an SQL command to the backend database. This is provided for backward compatibility with early versions of Clarion. We recommend using the property syntax to send SQL statements to the backend database.

Examples:

```
SEND(SQLFile,'SELECT field1,field2 FROM table1'      |
      & 'WHERE field1 > (SELECT max(field1)'        |
      & 'FROM table2')                               !Returns a result set you
                                                    ! get one row at a time
                                                    ! using NEXT(SQLFile)
```

```
SEND(SQLFile,'CALL GetRowsBetween(2,8)')           !Call stored procedure
```

```
SEND(SQLFile,'CREATE INDEX ON table1(field1 DESC)') !No result set
```

SQL does a good job of handling batch processing procedures such as: printing reports, displaying a screen full of table rows, or updating a group of table rows.

The SQL Accelerator drivers take full advantage of this when browsing a table or printing. However, they do not always use it to its best advantage with the Process template or in code which loops through a table to update multiple records. Therefore, when doing batch updates to a table, it can be much more efficient to execute an embedded SQL statement than to rely on the code generated by the Process template.

For example, to use PROP:SQL to increase all Salesman salaries by 10% you could:

```
SQLFile  FILE,DRIVER('Oracle'),NAME(SalaryFile)
Record   RECORD
SalaryAmount  PDECIMAL(5,2),NAME('JOB')
. .
CODE
SqlFile{PROP:SQL} = 'UPDATE SalaryFile SET '&|
                    'SALARY=SALARY * 1.1 WHERE JOB='S'''
```

The names used in the SQL statement are the SQL table names, not the Clarion field names.

PROP:SQLFilter

You can use PROP:SQLFilter to filter your VIEWS using native SQL code rather than Clarion code. This is only appropriate when using an SQL file driver (such as the ODBC, Scalable SQL, or Oracle drivers). If the first character of the PROP:SQLFilter expression is a plus sign (+), the PROP:SQLFilter expression is appended to any existing PROP:Filter expression and both are used. Omitting the plus sign replaces the existing PROP:Filter expression with the PROP:SQLFilter.

When you use PROP:SQLFilter, the SQL filter is passed directly to the server. As such it cannot contain the names of tables, variables, or functions that the server is not aware of; that is the filter expression must be valid SQL syntax with valid SQL table and column names. For example:

```
View{PROP:SQLFilter} = 'Date = TO_DATE(''01-MAY-1996'', 'DD-MON-YYYY'' )'
```

or

```
View{PROP:SQLFilter} = 'StrField LIKE ''AD%'''
```

Note that the SQL Accelerator incorporates the PROP:SQLFilter expression into the WHERE clause of a generated SELECT statement. The generated SELECT statement may reference one or more tables by aliases. If your filter also references tables (e.g., `Customer.Name < 'T'`), you must use the same alias names generated by the SQL Accelerator. By default, the SQL Accelerator uses the next letter of the alphabet as the alias name. For example, the Accelerator uses 'A' as the alias for the first table in the generated SELECT statement, then 'B' for the next table, and so on. You can use PROP:Alias to control the alias names generated by the SQL Accelerator. See *PROP:Alias* for more information.

Combining VIEW Filters and SQL Filters

When you use PROP:SQLFilter, the SQL filter may replace any filter specified for the VIEW, or it may be in addition to a filter specified for the VIEW. Prefix the SQL filter with a plus sign (+) to append the SQL filter to the existing VIEW filter. For example:

```
View{PROP:SQLFilter} = '+ StrField LIKE ''AD%'''
```

When you append the SQL filter by using the plus sign, the logical end result of the filtering process is (View Filter) AND (SQL Filter).

Omit the plus sign (+) to replace the Clarion filter with the SQL filter. When you replace the Clarion filter with the SQL filter by omitting the plus sign, the logical end result of the filtering process is simply (SQL Filter).

See Also: PROP:Filter for additional information.

PROP:SQLJoinExpression

You can use PROP:SQLJoinExpression to structure your VIEWS using native SQL code rather than Clarion code.

Note: Using PROP:SQLJoinExpression may hurt performance in some circumstances.

When you use PROP:SQLJoinExpression, the SQL join expression is passed directly to the server. As such it cannot contain the name of variables or functions that the server is not aware of; that is the join expression must be valid SQL syntax with valid SQL column names. For example:

```
View{PROP:SQLJoinExpression} = 'TO_DATE - FROM_DATE'
```

Combining VIEW Orders and SQL Orders

When you use PROP:SQLJoinExpression, the SQL join expression may replace any the join specified for the VIEW, or it may be in addition to the join specified for the VIEW. Prefix the SQL join with a plus sign (+) to concatenate the SQL join expression to the existing VIEW join expression. For example:

```
View{PROP:SQLOrder} = '+ TO_DATE - FROM_DATE'
```

When you concatenate the SQL join by using the plus sign, the result set contains first the Clarion joined values, then the SQL joined values.

Omit the plus sign (+) to replace the Clarion join expression with the SQL join expression.

See *PROP:JoinExpression* in the *Language Reference* for more information.

Tip: PROP:SQLJoinExpression only affects the JOIN portions of the VIEW declaration; it does not affect the PROJECT portions.

PROP:SQLOrder

You can use PROP:SQLOrder to sort your VIEWS using native SQL code rather than Clarion code.

Note: Using PROP:SQLOrder may hurt performance in some circumstances.

When you use PROP:SQLOrder, the SQL order is passed directly to the server. As such it cannot contain the name of tables, variables, or functions that the server is not aware of; that is the order expression must be valid SQL syntax with valid SQL column names. For example:

```
View{PROP:SQLOrder} = 'TO_DATE - FROM_DATE'
```

Note that the SQL Accelerator incorporates the PROP:SQLOrder expression into the ORDERBY clause of a generated SELECT statement. The generated SELECT statement may reference one or more tables by aliases. If your order expression also references tables (e.g., `Customer.Name < 'T'`), you must use the same alias names generated by the SQL Accelerator. By default, the SQL Accelerator uses the next letter of the alphabet as the alias name. For example, the Accelerator uses 'A' as the alias for the first table in the generated SELECT statement, then 'B' for the next table, and so on. You can use PROP:Alias to control the alias names generated by the SQL Accelerator. See *PROP:Alias* for more information.

Combining SQL Orders and VIEW Orders

When you use PROP:SQLOrder, the SQL order may replace any order specified for the VIEW, or it may be in addition to the order specified for the VIEW. Prefix the SQL order with a plus sign (+) to append the SQL order to the existing VIEW order. For example:

```
View{PROP:SQLOrder} = '+ TO_DATE - FROM_DATE'
```

When you append the SQL order by using the plus sign, the result set is ordered first by the Clarion order expression, then by the SQL order expression.

Omit the plus sign (+) to replace the Clarion order with the SQL order.

See *PROP:Order* in the *Language Reference* for more information.

ADO Database Driver

What is ADO?

ADO is a Microsoft technology, and stands for ActiveX Data Objects. It is a high-level programming interface used to access data in a database. ADO is designed as an easy-to-use application level interface to Microsoft's low-level data access interface, OLE DB.

ADO is automatically installed with Microsoft IIS as an Active X component. ADO is a common way to access a database from inside a web page (like an ASP page). For example, to connect to a database inside an ASP page:

1. Create an ADO connection to a database
2. Open the database connection
3. Create an ADO recordset
4. Open the recordset
5. Extract the data you need from the recordset
6. Close the recordset
7. Close the connection

The important thing to note here is the specific opening and closing of the database connection. Failure to specifically close an ADO connection can result in memory leakage.

ADO Requirements

The use of Clarion with ADO requires that you have installed the Microsoft Data Access Components (MDAC) interface, which is a free download from the Microsoft web site. You must have Version 2.62 or later installed.

The general flow of using the new ADO interface in Clarion:

1. Import your ADO tables using the Dictionary Editor.
2. Add the ADO Global Extension to your application
3. Add your various ADO procedures as necessary.

ADO Logging

The ADO Synchronizer supports trace logging.

Logging can only be turned on via the WIN.INI.

The section in the WIN.INI is called

```
[CWADOSYNCHRONISER]
```

Two possible settings in this section are:

```
Trace=0|1
```

```
TraceFile=filename
```

Trace must be set to 1 to turn on logging

TraceFile specifies the file you want to log to. If not supplied then the log is *ADOSYNCHRONISER.LOG*

The DrvTrace example application has been updated to support logging of the ADO Synchronizer.

This logging is active during the synchronization process with the Dictionary Editor. It is not to be confused with the normal trace logging that can be active in your application at runtime.

MSSQL Accelerator

MSSQL Accelerator Overview

MSSQL Server

For complete information on the MSSQL database system, please review Microsoft's SQL Server documentation.

MSSQL Accelerator

The MSSQL Accelerator is one of several SoftVelocity SQL Accelerator drivers. These SQL Drivers share a common code base and many common features such as SoftVelocity's unique, high speed buffering technology, common driver strings, and SQL logging capability. **See *SQL Accelerators* for information on these common features.**

The MSSQL Accelerator converts standard Clarion file I/O statements and function calls into optimized SQL statements, which it sends to the backend MSSQL server for processing. This means you can use the same Clarion code to access both MSSQL tables and other file systems such as TopSpeed files. It also means you can use Clarion template generated code with your SQL databases.

All the common behavior of all the SQL Accelerators is documented in the SQL Accelerators section. All behavior specific to this driver is noted here.

MSSQL Accelerator SQL Import Wizard--Login Dialog

Clarion's Dictionary Editor Import Wizard lets you import MSSQL table definitions into your Clarion Data Dictionary. When you select the MSSQL Accelerator from the driver drop-down list, the Import Wizard opens the **Login/Connection** dialog. The **Login/Connection** dialog collects the connection information for the MSSQL database.

Note: If you are using a **Trusted Connection (Integrated NT Security)**, you must establish a connection to the NT workstation running the MSSQL Server before you can connect to the MSSQL database and import table definitions. You can verify your connection by running the MSSQL ISQL_w Server utility installed with your MSSQL Client software.

Fill in the following fields in the **Login/Connection** dialog:

Servername

Select the workstation running the MSSQL database to import from. If the **Servername** list is empty, you may type in the name. See your DBA or network administrator for information on how the server is specified.

Username

For Standard Security, type your MSSQL Username. For Trusted Security (Integrated NT Security) no Username is required. See your server documentation or your DBA for information on applicable Usernames and security methods.

Password

For Standard Security, type your MSSQL Password. For Trusted Security (Integrated NT Security) no Password is required. See your server documentation or your DBA for information on applicable Passwords and security methods.

Database

Select the MSSQL database that contains the tables or views to import. If the **Database** list is empty, you may type in the name. See your server documentation or your DBA for information on database names.

Filter

Optionally, provide a filter expression to limit the list of tables and views to import. The filter expression queries the `dbo.sysobjects` table. The filter expression is limited to 1024 characters in length.

Tip: The filter is case sensitive, so type your filter value accordingly.

Following is a list of the column names (and their Clarion datatypes) you can reference in your filter expression. Generally, filtering on MSSQL system tables requires not only an intimate knowledge of the MSSQL system tables, but also of the MSSQL stored procedures. For example, to filter on table owner:

```
uid = user_id('DBO')
```

See your SQL server documentation for information on the MSSQL system tables and stored procedures.

name	CSTRING(31)
id	LONG
uid	SHORT
type	STRING(2)
userstat	SHORT
sysstat	SHORT
indexdel	SHORT
schema_ver	SHORT
refdate	STRING(8)
crdate	STRING(8)
version	LONG
deltrig	LONG
instrig	LONG
updtrig	LONG
seltrig	LONG
category	LONG
cache	SHORT

Next >

Press this button to open the Import Wizard's **Import List** dialog.

MSSQL Accelerator SQL Import Wizard--Import List Dialog

When you press the **Next >** button, the Import Wizard opens the **Import List** dialog. The **Import List** dialog lists the importable items.

Highlight the table or view whose definition to import, then press the **Finish** button to import. The Import Wizard adds the definition to your Clarion Data Dictionary, then opens the **File Properties** dialog to let you modify the default definition.

Import additional tables or views by repeating these steps. After all the items are imported, return to the Dictionary Editor where you can define relationships and delete any columns not used in your Clarion application. See *SQL Accelerators--Define Only the Fields You Use*.

Tip: You can use the Clarion Enterprise Edition Dictionary Synchronizer to import an entire database, including table relationships, in a single pass.

MSSQL Accelerator Connection Information

(and Driver Configuration--File Properties)

Typically, you add MSSQL support to your application by importing the table definitions into your Clarion Data Dictionary. The Import Wizard automatically fills in the **File Properties** dialog with default values based on the imported item. However, you can use the **Owner Name** field in the **File Properties** dialog to further configure the way the MSSQL Accelerator accesses the data.

The OWNER attribute for MSSQL takes the format:

```
server , <database> , <uid> , <pwd><;LANGUAGE=language><;APP=name><;WSID=name>
```

LANGUAGE

The language used by MSSQL Server.

APP

The name of the application.

WSID

The workstation ID. Typically, this is the network name of the computer on which the application resides.

See your MSSQL Server documentation for information on these settings.

Tip: Type an exclamation point (!) followed by a variable name in the Owner Name field to specify a variable rather than hard coding the OWNER attribute . For example: !GLO:SQLOwner.

MSSQL Accelerator Performance Considerations

The MSSQL Accelerator uses cursors. The MSSQL Server will not use an Index with a cursor, but it will use a Unique Constraint with a cursor. Therefore we recommend using Unique Constraints rather than Indexes wherever possible.

MSSQL Accelerator Calling a Stored Procedure

Stored procedures can return an output parameter and return results. These can only be returned if the file is opened in Read-Only mode (10H). The output parameters and return results are not available until all records have been retrieved by a SELECT statement.

```
PROGRAM
MAP
  CheckError(String) !Check for errorcodes
  CallProc(String) !Call Stored Procedure
END

MyFile FILE,DRIVER('MSSQL')
Record RECORD
c      LONG
      END
      END

Ret    LONG
In     SHORT(10)
Out    STRING(10)
CreateReq BYTE(FALSE)

CODE
BIND('Ret',Ret)
CheckError('BIND Ret')
BIND('Out',Out)
CheckError('BIND Out')
BIND('In',In)
CheckError('BIND In')

MyFile{PROP:SQL} = 'DROP TABLE SProctable'
MyFile{PROP:SQL} = 'CREATE TABLE SProctable (c INT)'

!Give MyFile initial data
OPEN(MyFile)
CheckError('Open')
MyFile.c=5
ADD(MyFile)
CheckError('Add 5')
MyFile.c=7
ADD(MyFile)
```

```

CheckError('Add 7')
MyFile.c=8
ADD(MyFile)
CheckError('Add 8')

!Initialize and Create Stored Procedures
MyFile{PROP:SQL} = 'DROP PROC SProc1'
MyFile{PROP:SQL} = 'DROP PROC SProc2'
MyFile{PROP:SQL} = 'DROP PROC SFunc1'
MyFile{PROP:SQL} = 'DROP PROC SFunc2'
MyFile{PROP:SQL} = 'DROP PROC SFunc3'
CallProc('CREATE PROC SFunc1 @input VARCHAR(10),@output VARCHAR(10) '& |
'OUTPUT AS SELECT @output = CHAR(ASCII('')+c) FROM SProctable ' & |
'WHERE c=7 ' & RETURN ASCII(@input) ')
CallProc('CREATE PROC SFunc2 @sin INT, @strin VARCHAR(10) AS ' & |
' SELECT c FROM SProctable RETURN @sin + ASCII(@strin)')
CallProc('CREATE PROC SFunc3 @input VARCHAR(10) AS ' & |
' RETURN ASCII(@input) ')
CallProc('CREATE PROC SProc1 @inp INT AS ' & |
' INSERT INTO SProctable values(@inp) ')
CallProc('CREATE PROC SProc2 @inp INT AS ' & |
' INSERT INTO SProctable values(@inp) ' & |
' SELECT c FROM SProctable')
CLOSE(MyFile)

!Open MyFile in Read-Only mode
OPEN(MyFile,10H)

!Call Stored Procedure passing input value using NORESULTCALL
!sets output parameter
CallProc('&Ret = NORESULTCALL SFunc3(''1'')')
IF Ret ~= VAL('1')
MESSAGE('Ret of NORESULTCALL SFunc3 =' & Ret)
END

!Call Stored Procedure passing input value, sets output parameter
CallProc('&Ret = CALL SFunc3(10)')
IF Ret ~= VAL('1')
MESSAGE('Ret of SFunc3 =' & Ret)
END

!Call Stored Procedure passing input value, no return values
CallProc('CALL SProc1(10)')

!Call Stored Procedure passing input value, return return code
CallProc('CALL SProc1(&in[IN])')

```

```
!Call Stored Procedure passing input value, return output parameter
CallProc('&Ret = CALL SFunc1(''1'',&out)')
IF Ret ~= VAL('1')
    MESSAGE('Ret of SFunc1 =' & Ret)
END
IF out ~= CHR(VAL('')+7)
    MESSAGE('out of SFunc1 =' & out)
END

!Call Stored Procedure passing input value, return return code
CallProc('CALL SProc2(&in[IN])')

NEXT(MyFile)
CheckError('Next SProc2')
!Call Stored Procedure passing input values, return output parameter
CallProc('&Ret = CALL SFunc2(7, '')')
LOOP WHILE ~ERRORCODE()
    NEXT(MyFile)
END
IF Ret ~= VAL('')+7
    MESSAGE('out of SFunc2 =' & out)
END
MESSAGE('Done')

!Check for errorcodes
CheckError PROCEDURE(Msg)
CODE
IF ERRORCODE()
IF ERRORCODE() = 90
    MESSAGE(Msg & ' ' & FILEERRORCODE() & ':' & FILEERROR())
ELSE
    MESSAGE(Msg & ' ' & ERRORCODE() & ':' & ERROR())
END
END

!CallProc calls the stored procedures using the PROP:SQL statement
CallProc PROCEDURE(Str)
CODE
MyFile{PROP:SQL} = Str
CheckError(Str)
```

MSSQL Accelerator Using Embedded SQL

Calling a Stored Procedure

Stored procedures can return an output parameter and return results. These can only be returned if the file is opened in Read-Only mode (10H). The output parameters and return results are not available until all records have been retrieved by a SELECT statement.

```
PROGRAM
MAP
  CheckError(STRING) !Check for errorcodes
  CallProc(STRING) !Call Stored Procedure
END

MyFile FILE,DRIVER('MSSQL')
Record RECORD
c      LONG
      END
      END

Ret      LONG
In       SHORT(10)
Out      STRING(10)
CreateReq BYTE(FALSE)

CODE
BIND('Ret',Ret)
CheckError('BIND Ret')
BIND('Out',Out)
CheckError('BIND Out')
BIND('In',In)
CheckError('BIND In')

MyFile{PROP:SQL} = 'DROP TABLE SProctable'
MyFile{PROP:SQL} = 'CREATE TABLE SProctable (c INT)'

!Give MyFile initial data
OPEN(MyFile)
CheckError('Open')
MyFile.c=5
ADD(MyFile)
CheckError('Add 5')
MyFile.c=7
ADD(MyFile)
CheckError('Add 7')
MyFile.c=8
```

```

ADD(MyFile)
CheckError('Add 8')

!Initialize and Create Stored Procedures
MyFile{PROP:SQL} = 'DROP PROC SProc1'
MyFile{PROP:SQL} = 'DROP PROC SProc2'
MyFile{PROP:SQL} = 'DROP PROC SFunc1'
MyFile{PROP:SQL} = 'DROP PROC SFunc2'
MyFile{PROP:SQL} = 'DROP PROC SFunc3'
CallProc('CREATE PROC SFunc1 @input VARCHAR(10),@output VARCHAR(10) '& |
'OUTPUT AS SELECT @output = CHAR(ASCII('')+c) FROM SProctable ' & |
'WHERE c=7 ' & RETURN ASCII(@input) ')
CallProc('CREATE PROC SFunc2 @sin INT, @strin VARCHAR(10) AS ' & |
' SELECT c FROM SProctable RETURN @sin + ASCII(@strin)')
CallProc('CREATE PROC SFunc3 @input VARCHAR(10) AS ' & |
' RETURN ASCII(@input) ')
CallProc('CREATE PROC SProc1 @inp INT AS ' & |
' INSERT INTO SProctable values(@inp) ')
CallProc('CREATE PROC SProc2 @inp INT AS ' & |
' INSERT INTO SProctable values(@inp) ' & |
' SELECT c FROM SProctable')
CLOSE(MyFile)

!Open MyFile in Read-Only mode
OPEN(MyFile,10H)

!Call Stored Procedure passing input value using NORESULTCALL
!sets output parameter
CallProc('&Ret = NORESULTCALL SFunc3(''1'')')
IF Ret ~= VAL('1')
MESSAGE('Ret of NORESULTCALL SFunc3 =' & Ret)
END

!Call Stored Procedure passing input value, sets output parameter
CallProc('&Ret = CALL SFunc3(10)')
IF Ret ~= VAL('1')
MESSAGE('Ret of SFunc3 =' & Ret)
END

!Call Stored Procedure passing input value, no return values
CallProc('CALL SProc1(10)')

!Call Stored Procedure passing input value, return return code
CallProc('CALL SProc1(&in[IN])')

!Call Stored Procedure passing input value, return output parameter
CallProc('&Ret = CALL SFunc1(''1'',&out)')
IF Ret ~= VAL('1')

```

```
    MESSAGE('Ret of SFunc1 =' & Ret)
END
IF out ~= CHR(VAL('')+7)
    MESSAGE('out of SFunc1 =' & out)
END

!Call Stored Procedure passing input value, return return code
CallProc('CALL SProc2(&in[IN])')

NEXT(MyFile)
CheckError('Next SProc2')
!Call Stored Procedure passing input values, return output parameter
CallProc('&Ret = CALL SFunc2(7, '')')
LOOP WHILE ~ERRORCODE()
    NEXT(MyFile)
END
IF Ret ~= VAL('')+7
    MESSAGE('out of SFunc2 =' & out)
END
MESSAGE('Done')
!Check for errorcodes
CheckError PROCEDURE(Msg)
CODE
IF ERRORCODE()
IF ERRORCODE() = 90
    MESSAGE(Msg & ' ' & FILEERRORCODE() & ':' & FILEERROR())
ELSE
    MESSAGE(Msg & ' ' & ERRORCODE() & ':' & ERROR())
END
END

!CallProc calls the stored procedures using the PROP:SQL statement
CallProc PROCEDURE(Str)
CODE
MyFile{PROP:SQL} = Str
CheckError(Str)
```

MSSQL Accelerator Driver Strings

There are switches or "driver strings" you can set to control the way your application creates, reads, and writes files with a specific driver. Driver strings are simply messages or parameters that are sent to the file driver at run-time to control its behavior. See *Common Driver Features--Driver Strings* for an overview of these runtime Database Driver switches and parameters.

Tip: A forward slash precedes all SQL Accelerator driver strings. The slash allows the driver to distinguish between driver strings and SQL statements sent with SEND.

In addition to the standard SQL Driver Strings, the MSSQL Accelerator supports the following Driver Strings:

HINT

You can tell MSSQL Accelerator to generate MSSQL hints by using the HINT driver string,

```
DRIVER ('MSSQL','/HINT=hint')
```

by using /HINT in the key name,

```
Key KEY(fieldlist),NAME('name /HINT=[&]hint')
```

with SEND,

```
SendReturn = SEND (file,' /HINT=[&]hint')
```

or with the PROP:Hint property,

```
file{PROP:Hint} = '[&]hint'
HintString = file{PROP:Hint}
```

The square brackets [] above are used to show that the ampersand (&) is optional.

You can either override the base hint or concatenate a hint. If the first character after the = in the KEY hint is an ampersand (&), MSSQL Accelerator concatenates the hint onto the FILE hint, otherwise it overrides the FILE hint.

If the first character after the = in the SEND hint is an ampersand (&) or the first character of a hint property is an ampersand, MSSQL Accelerator concatenates the hint onto the current hint (the FILE hint and the KEY hint), otherwise it overrides the FILE and KEY hint.

You can also use PROP:Hint to return the hint that is in use (or will be in use if called after a SET, but before the first NEXT or PREVIOUS statement.)

Example:

```
AFile DRIVER('MSSQL','/hint=COST')
AKey KEY(field),NAME('KeyName /HINT=&FIRST_ROWS')
SEND(AFile,'/HINT=FIRST_ROWS')
AFile{PROP:Hint} = 'FIRST_ROWS'
```

LOGONSCREEN (MSSQL Accelerator)

```
DRIVER('MSSQL', '/LOGONSCREEN = TRUE | FALSE ' )
```

```
[ AutoLogon" = ] SEND(file, '/LOGONSCREEN [ = TRUE | FALSE ]' )
```

See Also: PROP:LogonScreen.

GATHERATOPEN (MSSQL Accelerator)

```
DRIVER('MSSQL', '/GATHERATOPEN = TRUE | FALSE ' )
```

By default the driver delays gathering field information until it is required. However, some backends (like Sybase 11) perform poorly under these conditions. Setting GATHERATOPEN to TRUE forces the driver to gather most of the field information when the file is opened, which avoids a slowdown during program execution.

SAVESTOREDPROC (MSSQL Accelerator)

```
DRIVER('MSSQL', '/SAVESTOREDPROC= TRUE | FALSE ' )
```

```
[ SaveProc" = ] SEND(file, '/SAVESTOREDPROC [ = TRUE | FALSE ]' )
```

The MSSQL Accelerator executes SQL statements by creating temporary stored procedures on the server and executing them. By default (SAVESTOREDPROC=TRUE), these stored procedures remain on the server until connection to the server is dropped. To remove the procedures as soon as possible, set SAVESTOREDPROC=FALSE.

TRUSTEDCONNECTION (MSSQL Accelerator)

```
DRIVER('MSSQL', '/TRUSTEDCONNECTION = TRUE | FALSE ')
```

```
[ Trusted" = ] SEND(file, '/TRUSTEDCONNECTION [ = TRUE | FALSE ]')
```

By default (TRUSTEDCONNECTION=FALSE), the MSSQL Accelerator requests a standard connection to SQL Server. To connect using SQL Server integrated security, set TRUSTEDCONNECTION=TRUE.

Tip: To set the connection type, you must issue the TRUSTEDCONNECTION switch before the connection is made to the server.

MSSQL Accelerator Driver Properties

You can use Clarion's property syntax to query and set certain MSSQL Accelerator driver properties. In addition to the standard SQL Accelerator properties (see *SQL Accelerators--SQL Accelerator Properties*), the MSSQL Accelerator supports the following properties.

PROP:LogonScreen (MSSQL Accelerator)

PROP:LogonScreen sets or returns the toggle that determines whether the driver automatically prompts for logon information. By default (PROP:LogonScreen=True), the driver does display a logon window if no connect string is supplied. If set to False and there is no connect string, the OPEN(file) fails and FILEERRORCODE() returns '28000.' For example:

```
Afile FILE,DRIVER('MSSQL')
!file declaration with no userid and password
    END
CODE
AFile{PROP:LogonScreen}=True    !enable auto login screen
OPEN(Afile)
```

The automatic logon screen lets prompts for the following connection information. Consult your MSSQL documentation for more information on these prompts:

Server

Select the workstation running the MSSQL database to import from. If the **Server** list is empty, you may type in the name. See your DBA or network administrator for information on how the server is specified.

Logon ID

For Standard Security, type your MSSQL Username. For Trusted Security (Integrated NT Security) no Username is required. See your server documentation or your DBA for information on applicable Usernames and security methods.

Password

For Standard Security, type your MSSQL Password. For Trusted Security (Integrated NT Security) no Password is required. See your server documentation or your DBA for information on applicable Passwords and security methods.

Options

Press this button to enable the following prompts. See your MSSQL Server documentation for information on these prompts.

Database

Select the MSSQL database that contains the tables or views to access. If the **Database** list is empty, you may type in the name. See your server documentation or your DBA for information on database names.

Language

The language used by MSSQL Server.

Application Name

The name of the application.

WorkStation ID

The workstation ID. Typically, this is the network name of the computer on which the application resides.

MSSQL Accelerator Supported File Commands and Functions

File Attributes	Supported
CREATE	Y
DRIVER(<i>filetype</i> [, <i>driver string</i>])	Y
NAME	Y
ENCRYPT	N
OWNER(<i>password</i>)	Y ₁
RECLAIM	N
PRE(<i>prefix</i>)	Y
BINDABLE	Y
THREAD	Y
EXTERNAL(<i>member</i>)	Y
DLL(<i>flag</i>)	Y
OEM	N
File Structures	Supported
INDEX	Y
KEY	Y
MEMO	N
BLOB	Y
RECORD	Y
Index, Key, Memo Attributes	Supported
BINARY	N ₃
DUP	Y
NOCASE	Y
OPT	N
PRIMARY	Y

NAME		Y
Ascending Components		Y
Descending Components		Y
Mixed Components		Y
Field Attributes		Supported
DIM		N
OVER		Y
NAME		Y
File Procedures		Supported
BOF(<i>file</i>)	N	
BUFFER(<i>file</i>)		Y
BUILD(<i>file</i>)		Y
BUILD(<i>key</i>)		Y
BUILD(<i>index</i>)		Y ₃
BUILD(<i>index, components</i>)		Y ₃
BUILD(<i>index, components, filter</i>)		N
BYTES(<i>file</i>)		Y
CLOSE(<i>file</i>)		Y
COPY(<i>file, new file</i>)		N
CREATE(<i>file</i>)		Y
DUPLICATE(<i>file</i>)		Y
DUPLICATE(<i>key</i>)		Y
EMPTY(<i>file</i>)		Y
EOF(<i>file</i>)	N	
FLUSH(<i>file</i>)		N
LOCK(<i>file</i>)		N
NAME(<i>label</i>)		Y

OPEN(<i>file, access mode</i>)	Y
PACK(<i>file</i>)	N
POINTER(<i>file</i>)	N
POINTER(<i>key</i>)	N
POSITION(<i>file</i>)	N
POSITION(<i>key</i>)	Y
RECORDS(<i>file</i>)	Y
RECORDS(<i>key</i>)	Y
REMOVE(<i>file</i>)	Y
RENAME(<i>file, new file</i>)	N
SEND(<i>file, message</i>)	Y
SHARE(<i>file, access mode</i>)	Y
STATUS(<i>file</i>)	Y
STREAM(<i>file</i>)	N
UNLOCK(<i>file</i>)	N
Record Access	Supported
ADD(<i>file</i>)	Y
ADD(<i>file, length</i>)	N
APPEND(<i>file</i>)	Y
APPEND(<i>file, length</i>)	N
DELETE(<i>file</i>)	Y
GET(<i>file, key</i>)	Y
GET(<i>file, filepointer</i>)	N
GET(<i>file, filepointer, length</i>)	N
GET(<i>key, keypointer</i>)	N
HOLD(<i>file</i>)	N
NEXT(<i>file</i>)	Y

NOMEMO(<i>file</i>)	N
PREVIOUS(<i>file</i>)	Y
PUT(<i>file</i>)	Y
PUT(<i>file</i> , <i>filepointer</i>)	N
PUT(<i>file</i> , <i>filepointer</i> , <i>length</i>)	N
RELEASE(<i>file</i>)	N
REGET(<i>file</i> , <i>string</i>)	N
REGET(<i>key</i> , <i>string</i>)	Y
RESET(<i>file</i> , <i>string</i>)	N
RESET(<i>key</i> , <i>string</i>)	Y
SET(<i>file</i>)	Y
SET(<i>file</i> , <i>key</i>)	N
SET(<i>file</i> , <i>filepointer</i>)	N
SET(<i>key</i>)	Y
SET(<i>key</i> , <i>key</i>)	Y
SET(<i>key</i> , <i>keypointer</i>)	N
SET(<i>key</i> , <i>key</i> , <i>filepointer</i>)	N
SKIP(<i>file</i> , <i>count</i>)	Y
WATCH(<i>file</i>)	Y
Transaction Processing	Supported (see Note 2)
LOGOUT(<i>timeout</i> , <i>file</i> , ..., <i>file</i>)	Y ⁴
COMMIT	Y
ROLLBACK	Y
Null Data Processing	Supported
NULL(<i>field</i>)	Y
SETNULL(<i>field</i>)	Y
SETNONNULL(<i>field</i>)	Y

Notes:

- 1 We recommend using a variable password that is lengthy and contains special characters because this more effectively hides the password value from anyone looking for it. For example, a password like "dd...#\$...*&" is much more difficult to "find" than a password like "SALARY."

Tip

To specify a variable instead of the actual password in the Owner Name field of the File Properties dialog, type an exclamation point (!) followed by the variable name. For example: !MyPassword.

- 2 See also *PROP:Logout* in the *Language Reference*.
- 3 BUILD(index) sets internal driver flags to guarantee the driver generates the correct ORDER BY clause. The driver does not call the backend server.
- 4 Whether LOGOUT also LOCKs the table depends on the server's configuration for transaction processing. See your server documentation.

MSSQL Accelerator Synchronizer Server

Clarion's Enterprise Edition includes the MSSQL Synchronizer Server and the Data Dictionary Synchronizer. The Dictionary Synchronizer uses the Synchronizer Server to gather complete information about an MSSQL database.

The MSSQL Synchronizer Server is one of several used by the Dictionary Synchronizer. All the common behavior of all the SQL Accelerators is documented in the *SQL accelerators* section. All behavior specific to this driver is noted here.

MSSQL Accelerator Synchronizer Login Dialog

Clarion's Dictionary Synchronizer Wizard (Enterprise Edition) lets you import an entire MSSQL database definition into your Clarion Data Dictionary in a single pass. During this process, the Synchronizer Wizard opens an MSSQL login dialog. This dialog collects the connection information for the MSSQL database.

Note:

If you are using a Trusted Connection (Integrated NT Security), you must establish a connection to the NT workstation running the MSSQL Server before you can connect to the MSSQL database and import table definitions. You can verify your connection by running the MSSQL ISQL_w Server utility installed with your MSSQL Client software.

Fill in the following fields in the login dialog:

Host

Select the workstation running the MSSQL database to import from. If the **Host** list is empty, you may type in the name. See your DBA or network administrator for information on how the host is specified.

Database

Select the MSSQL database that contains the tables or views to import. If the **Database** list is empty, you may type in the name. See your server documentation or your DBA for information on database names.

Username

For Standard Security, type your MSSQL Username. For Trusted Security (Integrated NT Security) no Username is required. See your server documentation or your DBA for information on applicable Usernames and security methods.

Password

For Standard Security, type your MSSQL Password. For Trusted Security (Integrated NT Security) no Password is required. See your server documentation or your DBA for information on applicable Passwords and security methods.

Include System Files

Select this option to include system tables in the list of importable objects.

Exclude System Files

Select this option to exclude system tables from the list of importable objects.

Other Filter

Select this option to provide a filter expression to limit the list of tables and views to import. The filter expression queries the `dbo.sysobjects` table. The filter expression is limited to 1024 characters in length.

Tip**The filter is case sensitive, so type your filter value accordingly.**

Following is a list of the column names (and their Clarion datatypes) you can reference in your filter expression. Generally, filtering on MSSQL system tables requires not only an intimate knowledge of the MSSQL system tables, but also of the MSSQL stored procedures. For example, to filter on table owner:

```
uid = user_id('DBO')
```

See your SQL server documentation for information on the MSSQL system tables and stored procedures.

name	CSTRING(31)
id	LONG
uid	SHORT
type	STRING(2)
userstat	SHORT
sysstat	SHORT
indexdel	SHORT
schema_ver	SHORT
refdate	STRING(8)
crdate	STRING(8)
version	LONG
deltrig	LONG
instrig	LONG
updtrig	LONG

seltrig	LONG
category	LONG
cache	SHORT

ODBC Accelerator Driver

ODBC:Overview

The ODBC Accelerator Driver is one of several SoftVelocity SQL Accelerator drivers. These SQL Drivers share a common code base and many common features such as SoftVelocity's unique, high speed buffering technology, common driver strings, and SQL logging capability. **See *SQL Accelerator Drivers* for information on these common features.**

The ODBC Accelerator Driver converts standard Clarion file I/O statements and function calls into optimized SQL statements, which it sends to the backend SQL server for processing. This means you can use the same Clarion code to access both SQL tables and other file systems such as TopSpeed files. It also means you can use Clarion template generated code with your SQL databases.

The ODBC Accelerator Driver is slightly different from the other SQL drivers in that it is a *generic* SQL driver. It is not specific to a particular SQL server, but, in fact, works with any database or file system that supports the ODBC standard. This includes SQL systems such as AS400, Informix, MSSQL, Oracle, Scalable SQL, SQL Anywhere, Sybase, and many non-SQL systems as well (dBase, Excel, FoxPro, etc.). This chapter describes special issues and considerations that arise when using the ODBC Accelerator Driver to access data.

All the common behavior of all the SQL Accelerator drivers is documented in the *SQL Accelerator Drivers* chapter. All behavior specific to the ODBC driver is noted in this chapter.

Note:

You must have Microsoft ODBC 2.1 or higher to access 32-bit data sources with Clarion's Database Manager. Further, if the 32-bit data source is a Microsoft data source, you must also run Windows NT to access it with Clarion's Database Manager.

You can download Microsoft ODBC 2.1 from <ftp.sunet.se/ftp/pub/vendor/Microsoft/developr/ODBC/public/ODBC21.exe>.

What is ODBC?

ODBC (Open DataBase Connectivity) is a Windows "strategic interface" for accessing data from a variety of Relational Database Management Systems (RDBMS) across a variety of networks and platforms.

The ODBC standard was developed and is maintained by Microsoft, which publishes an ODBC Software Development Kit (SDK), geared for use with its Visual C++ product. ODBC support is another way in which Clarion provides an extensible platform for you to create applications.

ODBC Pros and Cons

Using ODBC offers the following advantages:

- † ODBC is an excellent choice in a Client-Server environment, especially if the Server is a native Structured Query Language (SQL) DBMS. It lets you add Client-Server support to your application, without having to do much more than choose a file driver. ODBC was specifically designed to create a non-vendor-specific method of connecting front end applications to back end services. With ODBC, the Server can handle much of the work, especially for SQL JOIN and PROJECT operations, thereby speeding up your application.
- † Existing ODBC drivers cover a great many types of databases. There are ODBC drivers available for databases for which Clarion may not have a native driver--for example, for Microsoft Excel and Lotus Notes files.
- † ODBC is already widespread. Major application suites such as Microsoft Office install ODBC drivers for file formats such as dBase and Microsoft Access. Keep in mind that many ODBC back end drivers have been updated and you should obtain the latest releases.
- † ODBC is platform independent. One of Microsoft's prime objectives in establishing ODBC was to support easier access to legacy systems, or corporate environments where data resides on diverse platforms or multiple DBMS's. As long as an ODBC driver and back end are available, it doesn't matter whether you use Microsoft's NetBEUI, SPX/IPX, DECNet or others; your application can connect to the DBMS and access the data.

Given that there are many drivers available, and that the standard was developed by the company that developed Windows, you might consider using ODBC as the driver of choice for *all* your Windows applications. Yet, when deciding whether to use an ODBC driver or a Clarion native database driver, you must also consider possible disadvantages:

- † ODBC adds a layer--the ODBC Driver Manager--between your application and the database. When accessing files on a local hard drive, this generally results in slower performance. The driver manager must translate the application's ODBC API call to an SQL statement before any data access occurs.

ODBC uses SQL to communicate with the back end database. Although this can be very efficient when communicating with Client/Server database engines, it is normally less efficient than direct record access when using a file system designed around single record access, such as xBase or Btrieve.
- † The ODBC Administrator manipulates the Windows registry in 32bit and the ODBC.INI in 16bit, adding complexity to systems that run under both 16-bit and 32-bit operation systems.
- † The information required by the ODBC database manager to connect to a data source varies from one ODBC driver to another. Unlike the selection of Clarion file drivers, where file operations are virtually transparent, you may need to do some work to gather the information required to use a particular ODBC driver. This chapter provides a few tips that might make it easier, and many ODBC drivers come with a Help (.HLP) file which documents special settings usually stored in ODBC.INI (16-bit only); but the burden is on *you* to solve any problems with third-party ODBC drivers.
- † ODBC is not included with Windows. When distributing your application, you'll need to install the ODBC drivers and the ODBC driver manager into the end user's system. This requires the ODBC SDK from Microsoft. In some cases, the back end server may have already provided a distribution kit which installs the ODBC driver on the workstation.
- † The normal Microsoft setup program that installs the ODBC driver manager adds an applet to the end user's Control Panel window for managing ODBC. It's very easy for an end user to use this tool to change the settings in the ODBC.INI file (16-bit only). The end user can unwittingly remove or modify the settings for the back end ODBC driver which would make it impossible for your application to connect to the data file.

Given the pros and cons, we recommend using the native Clarion file drivers when both a native driver and an ODBC driver exist for the same file format.

How ODBC Works

When you use ODBC to access data, four components must cooperate to make it work:

- † *Your application* calls the ODBC driver manager, and sends it the appropriate requests for data, with the ODBC API.

Clarion does this for you transparently, using the C60ODBX.DLL (32-bit) application extension. When hand-coding, be sure to include this library in the project. When distributing your application, be sure to deploy this file with your .EXE file (unless you produce a one-piece .EXE).

- † *The ODBC driver manager* receives the API calls, check the Windows Registry for information on the data source, then loads the ODBC "back-end" driver.

The actual "interface" to the driver manager is a file called ODBC32.DLL, which the Microsoft setup program places in the \Windows\System directory. This is the ODBC Administrator, which then loads other libraries to do its work.

- † *The ODBC "backend" driver* is another library (.DLL) which contains the executable code for accessing the data.

Various third-parties supply "backend" drivers. For example, Lotus Development Corp. supplies the ODBC driver for Lotus Notes. Microsoft Office distributes an ODBC SDK containing drivers for most of their database products.

- † *The data source* is either a data file (usually when ODBC is used for local data access), or a remote DBMS, such as an Oracle database.

The data source has a descriptive name; for example, "Microsoft Access Databases." The name serves as the section name in the ODBC.INI file.

The ODBC driver manager *must* know the exact data source name so that it can load the right driver to access the data. Therefore, it's vitally important that *you* know the precise data source name.

ODBC Data Types

Clarion Data Types													
ODBC Data Types	STRING	CSTRING	BYTE	SHORT	LONG	ULONG	SREAL	REAL	DECIMAL	PDECIMAL	DATE	TIME	
M	CHAR	C			
I	VARCHAR	.	C			
N	LONG VARCHAR	.	.										
	DECIMAL		.	.1	.1	.1		.4	.4		.		
	NUMERIC		.	.1	.1	.1		.4	.4	.			
C	SMALLINT		.	.3	C5	.3	.3						
O	INTEGER		.	.3	.3	C5	.3						
R	REAL		.					C	.3	.4	.4		
E	FLOAT		.					.3	.	.4	.4		
	DOUBLE PRECISION		.					.3	C	.4	.4		
E	BIT		.	.	.3	.3	.3						
X	TINYINT		.	C5	.3	.3	.3						
T	BIGINT		.		.3	.3	.3	.3	.3	.	.		
E	BINARY	.											
N	VARBINARY	.											
D	LONG VARBINARY	.											
E	DATE										C		
D	TIME											C	
	TIMESTAMP	.2											

Notes

- C The Clarion data type can be used to manipulate the ODBC data type. CREATE does create the ODBC data type.
- The Clarion type can be used to manipulate the ODBC data type, however, CREATE does NOT create the ODBC data type.
- 1 Clarion LONG, SHORT, and BYTE can be used with ODBC DECIMAL and NUMERIC data types if the ODBC field does not have any decimal places.
- 2 ODBC TIMESTAMP fields can be manipulated using a STRING(8) followed by a GROUP over it which contains only a DATE field and a TIME field.

Example:

```
TimeStampField STRING(8),NAME('TimeStampField')
TimeStampGroup GROUP,OVER(TimeStampField)
TimeStampDate DATE
TimeStampTime TIME
END
```

CREATE creates a TIMESTAMP field if you use a similar structure.

- 3 Some loss of precision may occur.
- 4 Rounding errors may occur.
- 5 CREATE attempts to create a TINYINT for a BYTE. If the backend does not support TINYINT, CREATE treats BYTE as a SHORT. CREATE attempts to create a SMALLINT for a SHORT. If the backend does not support SMALLINT, CREATE treats SHORT as a LONG. CREATE attempts to create an INTEGER for a LONG. If the backend does not support INTEGER, CREATE creates a decimal field.

Note:

Your backend database may contain data types that are not listed here. These data types are converted to ODBC data types by the backend database. Consult your backend database documentation to determine which ODBC data type is used.

Importing from ODBC Data Sources

Clarion's Dictionary Editor Import Wizard lets you import table definitions into your Clarion Data Dictionary.

When you select the ODBC Accelerator Driver from the driver drop-down list, the Import Wizard opens the **Data Sources** dialog. Select an existing Data Source, then press the **Next** button to import its definition.

If the data source is not defined, you can add it by pressing the **New** button, then following the ODBC instructions provided by the file system you wish to access.

When you have selected a data source, press the **Next** button to import its definition. This imports the ODBC table definition and opens the **File Properties** dialog, allowing you to modify file attributes, if you choose.

ODBC Connection Information and Driver Configuration--File Properties

Typically, you add SQL support to your application by importing the SQL or ODBC table, view, and synonym definitions into your Clarion Data Dictionary. The Import Wizard automatically fills in the **File Properties** dialog with default values based on the imported item. However, there are several fields in the **File Properties** dialog you can use to further configure the way the ODBC Accelerator Driver accesses the data. These **File Properties** fields, and their uses are described below.

Owner Name

Typically, the Import Wizard places the SQL database connection information (Host, Username, Password, etc.) in the **Owner Name** field.

Some backend databases may require additional connection information which you can supply in the **Owner Name** field. This information follows the password and is separated by semicolons, using the syntax: *keyword=value;keyword=value*.

For example, when accessing a Sybase database with the ODBC driver, this would appear as:

```
DataSource,UserID,Password,DATABASE=DataBaseName;APP=APPName
```

Consult your SQL Server's documentation for information on these keywords, their uses and effects.

ODBC Key Configuration--Key Properties

Typically, you add SQL support to your application by importing the SQL or ODBC table, view, and synonym definitions into your Clarion Data Dictionary. The Import Wizard automatically fills in the **Key Properties** dialog with default values based on the imported item.

ODBC Column Configuration--Field Properties

Typically, you add SQL support to your application by importing the SQL or ODBC table, view, and synonym definitions into your Clarion Data Dictionary. The Import Wizard automatically fills in the **Field Properties** dialog with default values based on the imported item. However, there are some fields in the **Field Properties** dialog you can use to further configure the way the SQL Accelerator Driver accesses the data. These **Field Properties** fields are described below.

External Name

| NOWHERE

Adding the NOWHERE switch to the **External Name** tells the ODBC driver to exclude the field from any WHERE clauses it sends to the backend server. This is necessary for certain backends when WATCH is in effect. Some backends do not allow certain data types in a where clause, but they fail to advise the ODBC Accelerator Driver of this restriction. The NOWHERE switch lets you manually advise of the restriction when WATCH causes the ODBC driver to generate. Not valid for Oracle driver.

| READONLY

Adding the READONLY switch to the **External Name** tells the ODBC driver not to insert the field when the record is added. This is necessary for certain back ends (such as Watcom) that do not allow auto incrementing key fields to be set to null. Some back ends do not allow auto incrementing key fields to be set to null, but they fail to advise the ODBC Accelerator Driver of this restriction. The READONLY switch lets you manually advise of the restriction. Not valid for Oracle driver.

| BINARY

Adding the BINARY switch to the **External Name** tells the ODBC driver to store the data in binary format. This is useful when storing images or non-printable characters. Valid only with STRING data types, and all SQL drivers (except Oracle).

Note: When adding attributes to the External Name feature, make sure to separate the fieldname from the appropriate switch (i.e., *fieldname / switch*). The spaces separating each value are required.

Debugging Your ODBC Application

When you use the ODBC Accelerator Driver, the ODBC Administrator can create a log file documenting all calls made by the ODBC Accelerator Driver. It includes the actual SQL statements made by the ODBC driver to the data source, and includes any errors posted. This administrator logging slows down your program considerably, so it should only be activated during testing. Additionally, the log file can grow to large proportions very quickly, so you should turn logging off and delete the log file after using it.

Besides "snooping" on the actual SQL statements generated by the driver, you can zero in on any errors. For example, if the application is unable to connect, you can open the log file, scroll to the bottom of the file, then work up until you find the word "SQLError."

See Microsoft's ODBC documentation (ODBCINST.HLP--*ODBC Options Dialog Box*) for instructions on using the ODBC Administrator logs.

ODBC:Supported Commands and Attributes

File Attributes	Supported
CREATE	Y
DRIVER(<i>filetype</i> [, <i>driver string</i>])	Y
NAME	Y
ENCRYPT	N
OWNER(<i>password</i>)	Y ₂
RECLAIM	N
PRE(<i>prefix</i>)	Y
BINDABLE	Y
THREAD	Y ₆
EXTERNAL(<i>member</i>)	Y
DLL(<i>flag</i>)	Y
OEM	N ₃
File Structures	Supported
INDEX	Y ₃
KEY	Y ₃
MEMO	N
BLOB	N
RECORD	Y
Index, Key, Memo Attributes	Supported
BINARY	N ₇
DUP	Y
NOCASE	Y
OPT	N

PRIMARY	Y
NAME	Y
Ascending Components	Y
Descending Components	Y
Mixed Components	Y

Field Attributes	Supported
-------------------------	------------------

DIM	N
OVER	Y
NAME	Y

File Procedures	Supported
------------------------	------------------

BOF(<i>file</i>)	N
BUFFER(<i>file</i>)	Y
BUILD(<i>file</i>)	Y
BUILD(<i>key</i>)	Y
BUILD(<i>index</i>)	Y ₈
BUILD(<i>index, components</i>)	Y ₈
BUILD(<i>index, components, filter</i>)	N
BYTES(<i>file</i>)	Y
CLOSE(<i>file</i>)	Y
COPY(<i>file, new file</i>)	N
CREATE(<i>file</i>)	Y
DUPLICATE(<i>file</i>)	Y
DUPLICATE(<i>key</i>)	Y
EMPTY(<i>file</i>)	Y
EOF(<i>file</i>)	N
FLUSH(<i>file</i>)	N

LOCK(<i>file</i>)	N
NAME(<i>label</i>)	Y
OPEN(<i>file, access mode</i>)	Y
PACK(<i>file</i>)	N
POINTER(<i>file</i>)	N
POINTER(<i>key</i>)	N
POSITION(<i>file</i>)	N
POSITION(<i>key</i>)	Y
RECORDS(<i>file</i>)	Y
RECORDS(<i>key</i>)	Y
REMOVE(<i>file</i>)	Y
RENAME(<i>file, new file</i>)	N
SEND(<i>file, message</i>)	Y
SHARE(<i>file, access mode</i>)	Y
STATUS(<i>file</i>)	Y
STREAM(<i>file</i>)	N
UNLOCK(<i>file</i>)	N
Record Access	Supported
ADD(<i>file</i>)	Y
ADD(<i>file, length</i>)	N
APPEND(<i>file</i>)	Y
APPEND(<i>file, length</i>)	N
DELETE(<i>file</i>)	Y
GET(<i>file, key</i>)	Y
GET(<i>file, filepointer</i>)	N
GET(<i>file, filepointer, length</i>)	N
GET(<i>key, keypointer</i>)	N

HOLD(<i>file</i>)	N
NEXT(<i>file</i>)	Y
NOMEMO(<i>file</i>)	N
PREVIOUS(<i>file</i>)	Y ₄
PUT(<i>file</i>)	Y
PUT(<i>file</i> , <i>filepointer</i>)	N
PUT(<i>file</i> , <i>filepointer</i> , <i>length</i>)	N
RELEASE(<i>file</i>)	N
REGET(<i>file</i> , <i>string</i>)	N
REGET(<i>key</i> , <i>string</i>)	Y
RESET(<i>file</i> , <i>string</i>)	N
RESET(<i>key</i> , <i>string</i>)	Y
SET(<i>file</i>)	Y ₄
SET(<i>file</i> , <i>key</i>)	N
SET(<i>file</i> , <i>filepointer</i>)	N
SET(<i>key</i>)	Y
SET(<i>key</i> , <i>key</i>)	Y
SET(<i>key</i> , <i>keypointer</i>)	N
SET(<i>key</i> , <i>key</i> , <i>filepointer</i>)	N
SKIP(<i>file</i> , <i>count</i>)	Y
WATCH(<i>file</i>)	Y
Transaction Processing	Supported (see Note 5)
LOGOUT(<i>timeout</i> , <i>file</i> , ..., <i>file</i>)	Y ₈
COMMIT	Y
ROLLBACK	Y
Null Data Processing	Supported
NULL(<i>field</i>)	Y

SETNULL(<i>field</i>)	Y
SETNONNULL(<i>field</i>)	Y

Notes:

- 1 The Clarion ODBC file driver supports the listed items, however, the underlying file system may not support all of these items.
- 2 We recommend using a variable password that is lengthy and contains special characters because this more effectively hides the password value from anyone looking for it. For example, a password like "dd...#\$...*&" is much more difficult to "find" than a password like "SALARY."

Tip

To specify a variable instead of the actual password in the Owner Name field of the File Properties dialog, type an exclamation point (!) followed by the variable name. For example: !MyPassword.

- 3 International sorting is assumed to be done by the underlying file system. As such the OEM attribute and the .ENV file are ignored.
- 4 PREVIOUS is not supported in file order.
- 5 See also *PROP:Logout* in the *Language Reference*.
- 6 THREADED files do not consume additional file handles for each thread that accesses the file.
- 7 BUILD(index) sets internal driver flags to guarantee the driver generates the correct ORDER BY clause. The driver does not call the backend server.
- 8 Whether LOGOUT also LOCKs the table depends on the server's configuration for transaction processing. See your server documentation..

ODBC:Driver Strings

There are switches or "driver strings" you can set to control the way your application creates, reads, and writes files with a specific driver. Driver strings are simply messages or parameters that are sent to the file driver at run-time to control its behavior. See *Common Driver Features--Driver Strings* for an overview of these runtime Database Driver switches and parameters.

Tip

A forward slash precedes all SQL driver strings. The slash allows the driver to distinguish between driver strings and SQL statements sent with the SEND function.

There are no Driver Strings specific to ODBC, the ODBC Accelerator Driver supports all the SQL Accelerator Driver Strings. See SQL Accelerator Driver Strings .

ODBC:Driver Properties

There are no driver properties specific to ODBC, the ODBC Accelerator Driver supports all the SQL Accelerator Driver properties. See SQL Accelerator Driver Properties .

Microsoft Access and ODBC

ODBC Driver that ships with Access 2.0

The ODBC driver that ships with Access 2.0 only works with other Microsoft Office applets. To get a general purpose driver that works with the Clarion ODCB Accelerator Driver, you need to purchase the ODBC Desktop Driver Kit 2.0 from Microsoft.

Oracle Accelerator

Overview

Oracle Server

For complete information on the Oracle database system, please refer to your Oracle documentation.

Oracle Accelerator

Oracle Accelerator is one of several SoftVelocity SQL Accelerators. These SQL drivers share a common code base and many common features such as SoftVelocity's unique, high speed buffering technology, common driver strings, and SQL logging capability. **See *SQL Accelerators* for information on these common features.**

The Oracle Accelerator converts standard Clarion file I/O statements and function calls into optimized SQL statements, which it sends to the backend Oracle server for processing. This means you can use the same Clarion code to access both Oracle tables and other file systems such as TopSpeed files. It also means you can use Clarion template generated code with your SQL databases.

All the common behavior of all the SQL Accelerators is documented in the SQL Accelerators section. All behavior specific to this driver is noted here.

SoftVelocity's Oracle Accelerator automatically works with Oracle versions 7.0 8i. At runtime, the driver initially tries to load the Oracle 8i DLLs. If the 8i DLLs are not available, it tries the 8.05 DLLs, 8.04 DLLs, 8.03 DLLs, 7.3 DLLs, and so on. See *Future Oracle Releases* for more information.

Note: Personal Oracle 8.0 only works with 32-bit programs.

Oracle Accelerator System Requirements

Hardware

You can run the Clarion development environment on any system that meets the minimum system requirements for Microsoft Windows 3.x, Windows 95™, or Windows NT 3.51.

Software

To *develop* Windows programs with Oracle Accelerator, you must have Oracle version 7.0 or higher; that is, you must have a licensed copy of Oracle's **ORAxWIN.DLL** (where x is the Oracle version number), plus any other DLLs it requires--typically Oracle's USDMEM.DLL and COREWIN.DLL. **These .DLLs must be in a directory that is in your system PATH.**

In addition, to *run* your *32-bit* programs with Oracle, you must have Oracle version 7.2 or higher; that is, you must have a licensed copy of Oracle's ORA72.DLL or higher. This DLL must also be in a directory that is in your system PATH.

Note:

Oracle 7.2 and higher (32-bit) does not automatically install all the .DLLs required by Clarion; you must specify Required Support Files for Windows (16-bit) in addition to the standard installation files. To do so, run SETUP.EXE from the WINDOWS directory on the Oracle CD.

You will not be able to define or import Oracle files in your Clarion data dictionary until the Oracle DLLs are installed in a directory that is in your system PATH.

Installing Oracle's Required Support Files

Oracle 7.2 and higher does not automatically install all the .DLLs required for Clarion with its standard install; you must install the Windows (16-bit) Required Support Files for your version of Oracle in addition to the standard installation files. You should follow Oracle's installation instructions for installing Required Support Files. We have found the following steps work best for Oracle 7.3:

1. Run SETUP.EXE from the WINDOWS directory on the Oracle CD.
2. Choose an install directory other than the primary Oracle directory.
3. Install the support files corresponding to your Oracle version only.

Oracle Accelerator Installation

To install the Oracle Accelerator file driver:

1. Install the required Oracle components.
See *System Requirements--Software*.
2. Run **A:SETUP** where A: is the drive letter of the drive containing the Oracle Accelerator install disk.

Follow the instructions on your screen. Install Oracle Accelerator to the directory that already contains Clarion.

The setup program installs Oracle Accelerator according to your selections. When everything is installed, the setup program offers to open the Oracle Accelerator on-line help file. This file contains late breaking information about the Oracle Accelerator file driver.
3. Please read the late breaking information.
When you have finished reading, close the file.
4. Register the Oracle Accelerator driver with the Clarion development environment.

Registering the Oracle Accelerator

You must register the Oracle Accelerator driver with the Clarion development environment before you can use it.

To register Oracle Accelerator:

1. Start the Clarion development environment.
2. Choose **Setup ▶ Database Driver Registry**.
3. Press the **Add** button.
4. Highlight C60ORA.DLL (by default, in the ..\BIN directory) in the list box, then press the **OK** button.
This registers the Oracle Accelerator driver.
5. Press the **OK** button.

Oracle Accelerator Table Import Wizard--Login Dialog

Clarion's Dictionary Editor Import Wizard lets you import Oracle table definitions into your Clarion Data Dictionary. When you select the Oracle Accelerator from the driver drop-down list, the Import Wizard opens the **Login/Connection** dialog. The **Login/Connection** dialog collects the connection information for the Oracle database.

IMPORTANT NOTES:

Before you can connect to the SQL database and import table definitions, the database must be started and must be accessible from your computer.

Only those indexes directly associated with the table as CONSTRAINTs are imported by the Oracle Accelerator driver. If you need more indexes, simply define them manually.

If the Oracle database INDEX flag is set to OFF, the Oracle Accelerator Import Wizard does not import CONSTRAINTS.

Fill in the following fields in the **Login/Connection** dialog:

Host

Select the Oracle host that contains the tables or views to import. If the **Host** list is empty, you may type in the host (a blank host specifies the Oracle 8 Personal database). See your DBA or network administrator for information on how the host is specified. For example, type 2: to connect to the local Personal Oracle (7.2 and earlier) database. X: prefixes an IPX host and TNS: prefixes a TCP/IP host.

Note: For Personal Oracle 8, leave the **Host** field blank.

Username

Type your Oracle Username. See your server documentation or your DBA for information on applicable Usernames.

Password

Type your Oracle Password. See your server documentation or your DBA for information on applicable Passwords.

Optionally, you type a complete connect string in the **Username** field using either of the following syntaxes:

```
username/password@Protocol:dbname
```

or

```
username@Protocol:dbname,password
```

For example type:

```
scott/tiger@2:production1
```

in the **Username** field. Or you may type just your username and the database name in the **Username** field, and type your password in the **Password** field. For example type:

```
scott@2:production1
```

in the **Username** field, then type

```
tiger
```

in the **Password** field. The Import Wizard displays the password as a series of asterisks. See your Oracle documentation for more information on Oracle connect string syntax.

Filter

Optionally, provide a filter expression to limit the list of tables and views to import. The filter expression queries the ALL_CATALOG view. For example the filter: OWNER='SCOTT' returns only the tables which have SCOTT as the OWNER. . The filter expression is limited to 1024 characters in length.

Tip: The filter is case sensitive, so type your filter value accordingly.

Following is a list of the ALL_CATALOG column names (and their Clarion datatypes) you can reference in your filter expression. See your Oracle documentation for more information on these columns.

OWNER	CSTRING(31)
TABLE_NAME	CSTRING(31)
TABLE_TYPE	CSTRING(12)

Next >

Press this button to open the Import Wizard's **Import List** dialog.

Oracle Accelerator Table Import Wizard--Import List Dialog

When you press the **Next >** button, the Import Wizard opens the **Import List** dialog. The **Import List** dialog lists the importable items.

Highlight the table, view, or synonym whose definition you wish to import, then press the **Finish** button to import. The Import Wizard adds the definition to your Clarion Data Dictionary, then opens the **File Properties** dialog to let you modify the imported definition.

Import additional tables by repeating these steps. After all the items are imported, return to the Dictionary Editor where you can define relationships and delete any columns not used in your Clarion application. See *SQL Accelerators--Define Only the Fields You Use*.

Tip: You can use the Enterprise Edition Dictionary Synchronizer to import an entire database, including relationships, in a single pass.

Oracle Accelerator Performance Considerations

See *SQL Accelerators--Performance Considerations* for more information on performance issues common to all SQL Accelerators, including Oracle Accelerator.

Oracle Accelerator Automatic Login Dialog

The Oracle Accelerator Login dialog lets the user specify Username, Password and Database.

In the **Database** drop-down list, select from previously selected Oracle hosts. The list of previously entered databases as well as the last UserID is stored in the Windows registry in the HKEY_CURRENT_USER/Software/SoftVelocity/Oracle tree as follows:

```
/Software/SoftVelocity/Oracle/UserID    the last UserID
/Software/SoftVelocity/Oracle/HostMRU  the last selected database
/Software/SoftVelocity/Oracle/HostCount number of databases in the list
/Software/SoftVelocity/Oracle/Host1    database name
/Software/SoftVelocity/Oracle/Host2    database name
/Software/SoftVelocity/Oracle/Hostn    database name
```

If the **Database** list is empty, you may simply type in the database name. For example, type 2: to connect to the local Personal database. The 2: indicates a local host; X: indicates an IPX host and TNS: indicates a TCP/IP host.

Alternatively, you may also supply a connect string (containing the database name) in the **Username** field. The Oracle connect string syntax is:

```
username/password@Protocol:dbname
```

or

```
username@Protocol:dbname,password
```

See your Oracle documentation for more information on Oracle connect string syntax.

If the **Username** field is not long enough, you may continue the entry in the **Password** field, because the Oracle Accelerator driver simply concatenates these fields and forwards their contents to the Oracle server.

Oracle Accelerator Generating Unique Key Values

For many database applications, you will want to automatically generate unique key values for your database records. An Oracle Sequence is simply a sequence number generator. You can select the next number from the Sequence whenever you add a new record.

Generally, we recommend using Oracle Sequences whenever possible to generate your unique key values. Sequences are more efficient because you never get a clash, and you need only two (2) database calls to add your new record.

Oracle Sequences

To use Oracle Sequences...

1. Define an Oracle Sequence for the auto incremented key.

See your Oracle documentation:

```
CREATE SEQUENCE CustomerSequence
INCREMENT BY 1
START WITH 1
NOMAXVALUE
MINVALUE 1
NOCYCLE
CACHE 30;
```

2. Declare a Clarion file to receive the sequence number from the Oracle Sequence like this:

```
CustomerSequence FILE,DRIVER( ' ORACLE' ),PRE(CUST) ,CREATE,THREAD
Record          RECORD,PRE( )
SequenceNo      LONG
                END
                END
```

3. Assign the incremented sequence number to your key field by embedding the following in the *WindowManager Method Executable Code Section - PrimeFields* embed point:

```
Access:CustomerSequence.Open
CustomerSequence{Prop:SQL}='Select CustomerSequence.Nextval from dual
IF ~Access:CustomerSequence.Next( )
Cust:CustNo=SequenceNo
END
```

where *Cust:CustNo* is the label of your auto-incremented key field in your Oracle data file.

4. Set the embedded code priority to 7500.

Oracle Accelerator Driver Strings

There are switches or "driver strings" you can set to control the way your application creates, reads, and writes files with a specific driver. Driver strings are simply messages or parameters that are sent to the file driver at run-time to control its behavior. **See *SQL Accelerators in the Programmer's Guide* for more information on SQL driver strings.**

In addition to the common SQL Accelerator driver strings, Oracle Accelerator supports the following driver strings as well.

HINT

You can tell Oracle Accelerator to generate Oracle hints by using the HINT driver string,

```
DRIVER ('Oracle','/HINT=hint')
```

by using /HINT in the key name,

```
Key KEY(fieldlist),NAME('name /HINT=[&]hint')
```

with SEND,

```
SendReturn = SEND (file,' /HINT=[&]hint')
```

or with the PROP:Hint property,

```
file{PROP:Hint} = '[&]hint'
HintString = file{PROP:Hint}
```

The square brackets [] above are used to show that the ampersand (&) is optional.

You can either override the base hint or concatenate a hint. If the first character after the = in the KEY hint is an ampersand (&), Oracle Accelerator concatenates the hint onto the FILE hint, otherwise it overrides the FILE hint.

If the first character after the = in the SEND hint is an ampersand (&) or the first character of a hint property is an ampersand, Oracle Accelerator concatenates the hint onto the current hint (the FILE hint and the KEY hint), otherwise it overrides the FILE and KEY hint.

You can also use PROP:Hint to return the hint that is in use (or will be in use if called after a SET, but before the first NEXT or PREVIOUS statement.)

Example:

```
AFile DRIVER('Oracle','/hint=COST')
AKey KEY(field),NAME('KeyName /HINT=&FIRST_ROWS')
SEND(AFile,'/HINT=FIRST_ROWS')
AFile{PROP:Hint} = 'FIRST_ROWS'
```

LOGON SCREEN

```
DRIVER('Oracle', '/LOGONSCREEN = TRUE | FALSE ')
```

LOGONSCREEN sets the toggle that determines whether the driver automatically prompts for logon information. By default, the driver displays a logon window if no connect string is supplied. If set to FALSE and there is no connect string, an ERRORCODE of 90 is returned.

The logon screen elements are contained within the file driver logic.

The end-user's ability to use the connect dialog will depend on the security surrounding the Oracle database.

PERSONAL

The Personal Oracle 7.1 Server behaves differently than other Oracle servers. When using Personal Oracle 7.1 you should inform Oracle Accelerator so it can tailor the generated SQL especially for Personal Oracle 7.1. For example:

```
DRIVER ('Oracle', '/PERSONAL')
```

or

```
SEND (Myfile, '/PERSONAL')
```

Note: The /PERSONAL switch is not required for Personal Oracle 7.2 (Personal Oracle for Windows 95).

Note: Personal Oracle 8.0 only works with 32-bit programs.

USEASYNCHRONOUSCALLS

The Oracle driver supports reading ahead asynchronously using the BUFFER statement. This may give performance gains when using this variation of the BUFFER statement. For example, you may have a large report where you need to read a lot of records and do some client side processing. It would improve performance to use the BUFFER's read ahead facility to get data in at the same time you are processing it. However, to enable this feature, all other commands will run slower (become asynchronous).

By default the asynchronous read ahead feature of the BUFFER statement is disabled. To enable this feature set **/USEASYNCHRONOUSCALLS=TRUE**

WHERE

In addition to WHERE driver string supported by all the SQL Accelerator drivers, Oracle Accelerator supports the following special WHERE driver string.

/Where in the FILE Definition

When a FILE declaration references more than one Oracle table, you must tell the Oracle server which columns link the tables together. A /WHERE in the FILE definition specifies the connecting fields between two or more Oracle tables. For example:

```
OrdBrowse FILE,DRIVER('ORACLE','/WHERE Orders.AccNum=Customer.AccNum'),|
          NAME('Orders,Customer'),PRE(Orb),BINDABLE,THREAD
OrdKey    KEY(-Orb:OrderNumber),NAME('OrdKey'),PRIMARY
Record    RECORD,PRE()
OrderNumber LONG,NAME('OrderNum')
AccountNumber LONG,NAME('Orders.AccNum')
ShipTo    STRING(32),NAME('ShipTo')
Name      STRING(31),NAME('Name')
          END
        END
```

Note: If you use the templates to generate your application, you will not need this technique. The templates automatically generate VIEWS when more than one table is referenced.

Oracle Accelerator Driver Properties

You can use Clarion's property syntax to query and set certain Oracle Accelerator driver properties. See *SQL Accelerators--SQL Accelerator Properties*.

Oracle Accelerator--Using Embedded SQL

You can use Clarion's property syntax (PROP:SQL) to send SQL and PL/SQL statements to the Oracle server within the normal execution of your program. For backward compatibility, you can also use SEND to send SQL and PL/SQL statements; however, we recommend using the property syntax.

See *SQL Accelerators* in the *Programmer's Guide* for more information on using embedded SQL.

PL/SQL

PL/SQL is Oracle's procedural language extension to Oracle's SQL language. Because PL/SQL statements are managed by the same engine that manages SQL statements, PL/SQL statements may be incorporated into your Clarion programs in the same manner as SQL statements. For example:

```

SQLFile FILE,DRIVER('Oracle'),NAME(SalaryFile)
Record RECORD
SalaryAmount PDECIMAL(5,2),NAME('JOB')
      END
      END

CODE
SqlFile{PROP:SQL} =
'DECLARE ' &|
'TempPhoneArea clarionclient.PhoneArea%type; ' &|
'CURSOR AreaCursor IS ' &|
'SELECT PhoneArea ' &|
'FROM ClarionClient ' &|
'WHERE PhoneArea = 305; ' &|
'BEGIN ' &|
'OPEN AreaCursor; ' &|
'LOOP ' &|
'FETCH AreaCursor INTO TempPhoneArea; ' &|
'EXIT WHEN AreaCursor%NOTFOUND; ' &|
'UPDATE ClarionClient ' &|
'SET PhoneArea = 954; ' &|
'END LOOP; ' &|
'CLOSE AreaCursor; ' &|
'COMMIT WORK; ' &|
'END;'

```

Calling a Stored Procedure:Oracle Accelerator

NORESULTCALL

For Oracle Accelerator, NORESULTCALL is required for stored procedures that do not return a result set. For example:

```
file{PROP:SQL} = 'NORESULTCALL GrantAccessProcedure'
```

DLL Coding Practices

Clarion applications that make use of DLLs must avoid calling certain file functions for Oracle tables in the constructors of static instances of classes declared in the EXE or DLLs, with the following exceptions:

- NAME(File)
- SEND(File, String)
- File{PROP:xxx} (get and set)
- Key{PROP:xxx} (get and set)

This is complete list. Any attempt to call any other file function in the constructors of static instances of classes can cause an "Oracle could not be found" error on attempt to load the Oracle client-side DLL.

ABC templates generate an instance of the DLLInitializer class for every DLL and provide embed points to enter custom code in the constructor. The constructor for the "data" DLL indirectly calls Init methods for every generated instance of the FileManager class. This method makes calls to the file functions from the list given above. Code entered in the provided embed points or added by 3rd party templates should not use calls to other file functions from DLLInitializer.Construct.

Oracle Accelerator Supported Commands and Attributes

File Attributes	Supported
CREATE	Y ₁
DRIVER(<i>filetype</i> [, <i>driver string</i>])	Y
NAME	Y
ENCRYPT	N
OWNER(<i>password</i>)	Y
RECLAIM	N
PRE(<i>prefix</i>)	Y
BINDABLE	Y
THREAD	Y
EXTERNAL(<i>member</i>)	Y
DLL(<i>flag</i>)	Y
OEM	Y ₂
File Structures	Supported
INDEX	Y
KEY	Y
MEMO	N
BLOB	Y
RECORD	Y
Index, Key, Memo Attributes	Supported
BINARY	N
DUP	Y
NOCASE	Y
OPT	Y
PRIMARY	Y

NAME		Y
Ascending Components		Y
Descending Components		Y
Mixed Components		Y
Field Attributes		Supported
DIM		N
OVER		Y
NAME		Y
File Procedures		Supported
BOF(<i>file</i>)	N	
BUFFER(<i>file</i>)		Y
BUILD(<i>file</i>)		Y
BUILD(<i>key</i>)		Y
BUILD(<i>index</i>)		Y ₃
BUILD(<i>index, components</i>)		Y ₃
BUILD(<i>index, components, filter</i>)		N
BYTES(<i>file</i>)		Y ₁₀
CLOSE(<i>file</i>)		Y
COPY(<i>file, new file</i>)		N
CREATE(<i>file</i>)		Y ₁
DUPLICATE(<i>file</i>)		Y
DUPLICATE(<i>key</i>)		Y
EMPTY(<i>file</i>)		Y
EOF(<i>file</i>)	N	
FLUSH(<i>file</i>)		N
LOCK(<i>file</i>)		N
NAME(<i>label</i>)		Y

OPEN(<i>file, access mode</i>)	Y
PACK(<i>file</i>)	N
POINTER(<i>file</i>)	N
POINTER(<i>key</i>)	N
POSITION(<i>file</i>)	Y ¹³
POSITION(<i>key</i>)	Y ¹¹
RECORDS(<i>file</i>)	Y
RECORDS(<i>key</i>)	Y ¹²
REMOVE(<i>file</i>)	Y
RENAME(<i>file, new file</i>)	N
SEND(<i>file, message</i>)	Y
SHARE(<i>file, access mode</i>)	Y
STATUS(<i>file</i>)	Y
STREAM(<i>file</i>)	N
UNLOCK(<i>file</i>)	N
Record Access	Supported
ADD(<i>file</i>)	Y
ADD(<i>file, length</i>)	N
APPEND(<i>file</i>)	Y ⁴
APPEND(<i>file, length</i>)	N
DELETE(<i>file</i>)	Y
GET(<i>file, key</i>)	Y
GET(<i>file, filepointer</i>)	Y ⁵
GET(<i>file, filepointer, length</i>)	N
GET(<i>key, keypointer</i>)	N
HOLD(<i>file</i>)	Y ⁶
NEXT(<i>file</i>)	Y

NOMEMO(<i>file</i>)	N
PREVIOUS(<i>file</i>)	Y ₇
PUT(<i>file</i>)	Y
PUT(<i>file</i> , <i>filepointer</i>)	N
PUT(<i>file</i> , <i>filepointer</i> , <i>length</i>)	N
RELEASE(<i>file</i>)	N
REGET(<i>file</i> , <i>string</i>)	Y ₈
REGET(<i>key</i> , <i>string</i>)	Y ₈
RESET(<i>file</i> , <i>string</i>)	N
RESET(<i>key</i> , <i>string</i>)	Y ₉
SET(<i>file</i>)	Y
SET(<i>file</i> , <i>key</i>)	N
SET(<i>file</i> , <i>filepointer</i>)	N
SET(<i>key</i>)	Y
SET(<i>key</i> , <i>key</i>)	Y
SET(<i>key</i> , <i>keypointer</i>)	N
SET(<i>key</i> , <i>key</i> , <i>keypointer</i>)	N
SKIP(<i>file</i> , <i>count</i>)	Y
WATCH(<i>file</i>)	Y
Transaction Processing	Supported
LOGOUT(<i>timeout</i> , <i>file</i> , ..., <i>file</i>)	Y
COMMIT	Y
ROLLBACK	Y
Null Data Processing	Supported
NULL(<i>field</i>)	Y
SETNULL(<i>field</i>)	Y ₁₃
SETNONNULL(<i>field</i>)	Y

Notes:

- 1 CREATE(*file*) does not work for every data type. See *Supported Data Types* for more information.
- 2 Adding the OEM attribute causes the driver to generate calls to NLSSORT for string fields in a sort sequence (either key components or PROP:Order components).
- 3 The BUILD(*dynamic index*) and BUILD(*index*) statements do not perform any disk action. They only initialize internal Oracle driver structures to track key order access and allow SELECT statements to be built when you issue SET(*key*) or SET(*key,key*) statements referencing the *index*.
- 4 The APPEND statement behaves identically to the ADD statement, that is, keys are updated by the APPEND statement.
- 5 The GET(*file, filepointer*) statement is unsupported for all values of *filepointer* except *filepointer* = 0. In this case, the record position is cleared and ERRORCODE 35 is returned.
- 6 Apart from the holding records, the HOLD statement has another use. Normally, the driver will not reread the record when you execute a RESET/NEXT to the current record. Executing a HOLD statement before the RESET/NEXT forces the driver to reread the record from disk.
- 7 You can't execute a PREVIOUS after a SET(*file*) statement. You can only examine the *file* in a forward order.
- 8 The REGET statement only works if you have a unique key defined for the file
- 9 The RESET(*key,position*)/NEXT(*file*) statement sequence is optimized to retrieve the record from the driver's internal buffer if the code is resetting to the current record. To force the driver to reread the record from disk, execute a HOLD statement before the RESET/NEXT sequence. This optimization is not in effect within a transaction frame.
- 10 The BYTES(*file*) function returns the number of records in the file or the number of bytes in the last record accessed. Following an OPEN statement, the BYTES function returns the number of records in the *file*. After the file has been accessed by GET, NEXT, ADD, or PUT, the BYTES function returns the size of the last *record* accessed.
- 11 The POSITION(*key*) function returns (1 + size of the *key* components + the size of the components of the file's primary key). This formula is true even if the first unique key is the same *key* you are positioning on. If no primary key is defined, then the first unique key is considered the primary key.

If there is no unique key, POSITION(*key*) returns 1 + size of the *key* components. In this case RESET(*key*) will reposition to the first occurrence of the key value, since there is no way of uniquely identifying a record. Therefore, the RESET may position on a different record.

- 12 The RECORDS(*key*) function returns the number of UNIQUE occurrences of the first element of the *key*. This is the same as doing an SQL statement of:
`SELECT COUNT (DISTINCT key_field1) FROM table`
- 13 SETNULL(*field*) clears the contents of the field.
- 14 The returned POSITION can only be used with REGET(*file,position*) and only for unique keys.

Oracle Accelerator Data Types

The following table matches Clarion data types to their corresponding Oracle data types.

Tip: Generally, you should not have to do any manual matching of data types. Rather, you simply import file definitions from your Oracle database into your Clarion data dictionary. The Oracle Accelerator driver automatically selects the proper data types (see *Importing Oracle Files to a Data Dictionary*).

<u>Oracle data type</u>	<u>Clarion data type</u>
CHAR	STRING
VARCHAR2	CSTRING
NUMBER	REAL
NUMBER(n,p)	PDECIMAL
NUMBER(n,0)	BYTE ₁ ,SHORT ₂ ,USHORT ₃ ,LONG ₄
LONG	STRING+GROUP+SHORT ₅
LONG RAW	STRING+GROUP+SHORT ₅
DATE	DATE or STRING+DATE+TIME ₆
RAW	STRING
ROWID	STRING(18) ₇

Data Type Notes:

- 1 CREATE will create a NUMBER(3,0).
- 2 CREATE will create a NUMBER(5,0).
- 3 CREATE will create a NUMBER(5,0).
- 4 CREATE will create a NUMBER(10,0).
- 5 Clarion can access Oracle LONG and LONG RAW data types by using a STRING and a GROUP overlaying the STRING. The GROUP consists of a SHORT and a STRING. The SHORT holding the length of the total data (including the length of the SHORT). For example:

```

Comments      STRING(1024),NAME('COMMENTS') !Oracle LONG field
COMMENTS_GROUP  GROUP,OVER(Comments)
COMMENTS_LENGTH  USHORT
COMMENTS_DATA    STRING(1022)
                END

```

You cannot use the CREATE statement to create rows of type LONG or LONG RAW.

- 6 Clarion can access Oracle DATE data types by using either a DATE or a STRING with a GROUP overlaying the STRING.

If you use a Clarion DATE field, the TIME component of the field is not readable and is set to 0 when writing the field. You may use a CREATE statement to create the table. For example:

```
StartDate    DATE,NAME('START_DATE')    !Oracle DATE field
```

If you use a Clarion STRING with an overlaid GROUP, the GROUP consists of a DATE and a TIME field. You may not use a CREATE statement to create the table. For example:

```
OraDate      STRING(8),NAME('START_DATE') !Oracle DATE field
StartDate_Group GROUP,OVER(OraDate)
StartDate    DATE
StartTime    TIME
END
```

- Tip:** Your Clarion application should generally reference the DATE field (StartDate), and should not reference the STRING field (OraDate) or the GROUP field (StartDate_Group).

However, if the Oracle date stamp is part of the key, you must include the STRING field (not the DATE field) as a key component in your Clarion data dictionary.

- 7 Oracle ROWID data types are read and written as a STRING(18) of formatBBBBBBBBB.RRRR.FFFF. Where B, R, and F are hexadecimal numbers representing block, row, and file number respectively. See your Oracle documentation for more information.

CREATE will not create a ROWID row.

Oracle Accelerator Troubleshooting

Clarion Won't Accept Oracle File Driver

Clarion's Dictionary Editor allows you to select the Oracle file driver only if Oracle is installed on your machine. That is, the Oracle DLLs must be installed in a directory in your path before Clarion's Dictionary Editor will recognize the Oracle driver. Attempting to run the Oracle example program will produce error windows that tell you which DLLs are missing.

Oracle Not Available (-1034)

If you receive this error (Oracle error number -1034), make sure the Oracle server is properly installed and is available to the client.

Unable to allocate memory on user side (-1019)

This message may indicate some of the required Oracle DLLs are not installed to a directory in your system path. See System Requirements--Software.

Unable to spawn new ORACLE (-9352)

This message may indicate the Oracle database is not started. Start the Oracle database and retry.

Could Not Log On:Oracle Accelerator

This message may indicate an invalid username, password, or servername. It may also indicate that the Oracle server is not installed or is otherwise not available. Make sure the Oracle server is properly installed and is available to the client.

Invalid Field Type Descriptor: Oracle Accelerator

The Clarion error: *Invalid Field Type Descriptor* is generated at runtime if you supply a field name in the field's NAME attribute that does not match any field name in the Oracle table. By turning logging on (see /LOGFILE) you can re-run your program and receive a list of valid field names in the log file. Use the dictionary import facility to import the field descriptions into your Clarion data dictionary to avoid this problem.

Unexpected End of SQL Command (-921)

If you receive this error (Oracle error number -921), make sure that the SQL Select statement selects the same number of columns as the receiving Clarion FILE structure declares.

This error may also occur when a CREATE statement generates incorrect SQL statements when the last field is OVER a previous field. Change the file layout so the last field declared is not declared OVER a previous field.

File Not Found:Oracle Accelerator

If you receive this error (or Oracle error number 942--Table or View Does Not Exist), check that the file name you are passing to the Oracle driver is valid. The most common occurrence of this error is when the name of the Oracle table is not correct. If the table is owned by another user, you must explicitly identify the owner as follows:

```
owner.table
```

Error 47:Oracle Accelerator

An Error 47 indicates there is a field defined in your Clarion data dictionary that does not exist on the Oracle server. To identify the field, use /LOGFILE.

Internal Error 02: WSLDIAL:Oracle Accelerator

Use the **Project Editor** dialog to add the ..\LIBSRC\ORALOON.RSC file to your application's **Library, object and resource files**.

To add resource files to your project:

1. From the Application Tree dialog, press the **Project** button.
2. Highlight **Library, object and resource files**, then CLICK on the **Add File** button.
3. Navigate to the ..\LIBSRC folder, then select the resource file (ORALOON.RSC) from the Windows File dialog.
4. Press the **OK** button to return to the **Project Editor** dialog.

No Interface Driver Connected(-03121)

This indicates you have not entered a correct database name. Be sure to enter the correct database name in the **Host** or **Database** field.

Scalable SQL Accelerator Driver

Scalable SQL Overview

The Scalable SQL Server

For complete information on the Scalable SQL database system, please review Pervasive Software's documentation.

Scalable SQL requires that the 16-bit ODBC support files are also installed.

The Scalable SQL Driver

The Scalable SQL Driver is one of several SoftVelocity SQL Accelerator drivers. These SQL Drivers share a common code base and many common features such as SoftVelocity's unique, high speed buffering technology, common driver strings, and SQL logging capability. **See SQL Accelerator Drivers for information on these common features.**

The Scalable SQL Driver converts standard Clarion file I/O statements and function calls into optimized SQL statements, which it sends to the backend Scalable SQL server for processing. This means you can use the same Clarion code to access both Scalable SQL tables and other file systems such as TopSpeed files. It also means you can use Clarion template generated code with your SQL databases.

All the common behavior of all the SQL Accelerator drivers is documented in the SQL Drivers section. All behavior specific to the Scalable SQL driver is noted in this chapter.

Scalable SQL Import Wizard--Login Dialog

Clarion's Dictionary Editor Import Wizard lets you import Scalable SQL table definitions into your Clarion Data Dictionary. When you select the Scalable SQL Accelerator Driver from the driver drop-down list, the Import Wizard opens the **Login/Connection** dialog. The **Login/Connection** dialog collects the connection information for the Scalable SQL database.

Note:

Before you can connect to the SQL database and import table definitions, the database must be started and must be accessible from your computer.

Fill in the following fields in the **Login/Connection** dialog:

Database Name

Select the Scalable SQL database that contains the tables to import. If the **Database Name** list is empty, you may type in the name.

See your server documentation for information on how the database is specified. The specification may depend on where the database server is located (remote or local), and on the network protocol (TCP/IP, IPX, etc.) used to access it.

DDF Directory

Press the **Browse** button to select the pathname or directory containing the database DDF files.

Database Directory

Press the **Browse** button to select the pathname or directory containing the database.

Note:

Specify either the Database Name or the DDF directory, but not both.

Owner Names

Optionally, type a comma separated list of names the Scalable SQL driver tries when opening encrypted Btrieve files. If a name contains a comma or space, it must be surrounded by single quotes.

Refresh table list

Check this box to refresh the list of tables to import when you press the **Next >** button. Clear the box to improve performance when the database is likely to be unchanged between imports.

Disconnect after Import or Cancel

Check this box to disconnect from the server after importing the (last) definition. Generally, you should clear this box when importing multiple definitions in order to maintain your connection to the server between imports.

Next >

Press this button to open the Import Wizard's **Import List** dialog.

Scalable SQL Import Wizard--Import List Dialog

When you press the **Next >** button, the Import Wizard opens the **Import List** dialog. The **Import List** dialog lists the importable items.

Highlight the table whose definition to import, then press the **Finish** button to import. The Import Wizard adds the definition to your Clarion Data Dictionary, then opens the **File Properties** dialog to let you modify the default definition.

Import additional tables by repeating these steps. After all the items are imported, return to the Dictionary Editor where you can define relationships and delete any columns not used in your Clarion application. See *SQL Accelerator Drivers--Define Only the Fields You Use*.

Scalable SQL Connection Information and Driver Configuration--File Properties

Typically, you add Scalable SQL support to your application by importing the table definitions into your Clarion Data Dictionary. The Import Wizard automatically fills in the **File Properties** dialog with default values based on the imported item. However, you can use the **Owner Name** field in the **File Properties** dialog to further configure the way the Scalable SQL Driver accesses the data.

Scalable SQL allows some information in addition to the database identification in the **Owner Name** field. This information appears alternatively as:

```
Database[ ,Owners;Switches ]
```

or

```
DDF=DDFPath[ |Datapath][ ,Owners;Switches ]
```

Where *Database* is the name of a Scalable SQL database. *DDFPath* is a path to a set of DDF (Btrieve data dictionary) files. *Datapath* is the path to the corresponding data files. If omitted, *Datapath* defaults to *DDFPath*. *Owners* is a comma separated list of names to try when opening encrypted Btrieve files. If a name contains a comma or space, it must be surrounded by single quotes. *Switches* is a semicolon separated list of assignments. Valid switches are:

```
CREATEDDF=[ 0 | 1 | 2 ]
```

Where *0* creates a new DDF file, *1* replaces the existing DDF file, and *2* removes the existing DDF File.

Note:

The **CREATEDDF** switch is provided primarily for use during initial installation to allow you to build new DDF files. You should never use this switch for existing databases.

PROP:MaxStatements

PROP:MaxStatements sets or returns the maximum amount of SQL statements that can be generated (open) on a single connection. A connection must be active before implementing this property.

```
number = file{PROP:MaxStatements} !return allowable  
file{PROP:MaxStatements} = 32767 !set allowable
```

Scalable SQL:Supported Commands and Attributes

File Attributes	Supported
CREATE	Y
DRIVER(<i>filetype</i> [, <i>driver string</i>])	Y
NAME	Y
ENCRYPT	N
OWNER(<i>password</i>)	Y ₁
RECLAIM	N
PRE(<i>prefix</i>)	Y
BINDABLE	Y
THREAD	Y
EXTERNAL(<i>member</i>)	Y
DLL(<i>flag</i>)	Y
OEM	N
File Structures	Supported
INDEX	Y
KEY	Y
MEMO	N
BLOB	Y
RECORD	Y
Index, Key, Memo Attributes	Supported
BINARY	N ₃
DUP	Y
NOCASE	Y
OPT	N

PRIMARY		Y
NAME		Y
Ascending Components		Y
Descending Components		Y
Mixed Components		Y
Field Attributes		Supported
DIM		N
OVER		Y
NAME		Y
File Procedures		Supported
BOF(<i>file</i>)	N	
BUFFER(<i>file</i>)		Y
BUILD(<i>file</i>)		Y
BUILD(<i>key</i>)		Y
BUILD(<i>index</i>)		Y ₃
BUILD(<i>index, components</i>)		Y ₃
BUILD(<i>index, components, filter</i>)		N
BYTES(<i>file</i>)		Y
CLOSE(<i>file</i>)		Y
COPY(<i>file, new file</i>)		N
CREATE(<i>file</i>)		Y
DUPLICATE(<i>file</i>)		Y
DUPLICATE(<i>key</i>)		Y
EMPTY(<i>file</i>)		Y
EOF(<i>file</i>)	N	
FLUSH(<i>file</i>)		N

LOCK(<i>file</i>)	N
NAME(<i>label</i>)	Y
OPEN(<i>file, access mode</i>)	Y
PACK(<i>file</i>)	N
POINTER(<i>file</i>)	N
POINTER(<i>key</i>)	N
POSITION(<i>file</i>)	N
POSITION(<i>key</i>)	Y
RECORDS(<i>file</i>)	Y
RECORDS(<i>key</i>)	Y
REMOVE(<i>file</i>)	Y
RENAME(<i>file, new file</i>)	N
SEND(<i>file, message</i>)	Y
SHARE(<i>file, access mode</i>)	Y
STATUS(<i>file</i>)	Y
STREAM(<i>file</i>)	N
UNLOCK(<i>file</i>)	N
Record Access	Supported
ADD(<i>file</i>)	Y
ADD(<i>file, length</i>)	N
APPEND(<i>file</i>)	Y
APPEND(<i>file, length</i>)	N
DELETE(<i>file</i>)	Y
GET(<i>file, key</i>)	Y
GET(<i>file, filepointer</i>)	N
GET(<i>file, filepointer, length</i>)	N
GET(<i>key, keypointer</i>)	N

HOLD(<i>file</i>)	N
NEXT(<i>file</i>)	Y
NOMEMO(<i>file</i>)	N
PREVIOUS(<i>file</i>)	Y
PUT(<i>file</i>)	Y
PUT(<i>file</i> , <i>filepointer</i>)	N
PUT(<i>file</i> , <i>filepointer</i> , <i>length</i>)	N
RELEASE(<i>file</i>)	N
REGET(<i>file</i> , <i>string</i>)	N
REGET(<i>key</i> , <i>string</i>)	Y
RESET(<i>file</i> , <i>string</i>)	N
RESET(<i>key</i> , <i>string</i>)	Y
SET(<i>file</i>)	Y
SET(<i>file</i> , <i>key</i>)	N
SET(<i>file</i> , <i>filepointer</i>)	N
SET(<i>key</i>)	Y
SET(<i>key</i> , <i>key</i>)	Y
SET(<i>key</i> , <i>keypointer</i>)	N
SET(<i>key</i> , <i>key</i> , <i>filepointer</i>)	N
SKIP(<i>file</i> , <i>count</i>)	Y
WATCH(<i>file</i>)	Y
Transaction Processing	Supported (see Note 2)
LOGOUT(<i>timeout</i> , <i>file</i> , ..., <i>file</i>)	Y ⁴
COMMIT	Y
ROLLBACK	Y
Null Data Processing	Supported
NULL(<i>field</i>)	Y

SETNULL(<i>field</i>)	Y
SETNONNULL(<i>field</i>)	Y

Notes:

- 1 We recommend using a variable password that is lengthy and contains special characters because this more effectively hides the password value from anyone looking for it. For example, a password like "dd...#\$...*&" is much more difficult to "find" than a password like "SALARY."

Tip

To specify a variable instead of the actual password in the Owner Name field of the File Properties dialog, type an exclamation point (!) followed by the variable name. For example: !MyPassword.

- 2 See also *PROP:Logout* in the *Language Reference*.
- 3 BUILD(index) sets internal driver flags to guarantee the driver generates the correct ORDER BY clause. The driver does not call the backend server.
- 4 Whether LOGOUT also LOCKs the table depends on the server's configuration for transaction processing. See your server documentation.

SQLAnywhere Accelerator

SQLAnywhere Accelerator Overview

SQLAnywhere Server

For complete information on the SQLAnywhere database system, please review Sybase's SQLAnywhere documentation.

SQLAnywhere Accelerator

The SQLAnywhere Accelerator is one of several SoftVelocity SQL Accelerators. These SQL Accelerators share a common code base and many common features such as SoftVelocity's unique, high speed buffering technology, common driver strings, and SQL logging capability. **See *SQL Accelerators* for information on these common features.**

The SQLAnywhere Accelerator converts standard Clarion file I/O statements and function calls into optimized SQL statements, which it sends to the backend SQLAnywhere server for processing. This means you can use the same Clarion code to access both SQLAnywhere tables and other file systems such as TopSpeed files. It also means you can use Clarion template generated code with your SQL databases.

All the common behavior of all the SQL Accelerators is documented in the *SQL accelerators* section. All behavior specific to this driver is noted here.

Start the SQLAnywhere Client

Clarion's Dictionary Synchronizer Wizard (Enterprise Edition) lets you import an entire SQLAnywhere database definition into your Clarion Data Dictionary in a single pass. Before you can connect to the SQLAnywhere database and import table definitions, you must start the database client software.

Note: Before you can connect to the SQLAnywhere database and import table definitions, you must start the database client software.

If you have not started the client software, Clarion issues the unable to start database engine message.

SQLAnywhere Accelerator SQL Import Wizard--Login Dialog

Clarion's Dictionary Editor Import Wizard lets you import SQLAnywhere table definitions into your Clarion Data Dictionary. When you select the SQLAnywhere Accelerator from the driver drop-down list, the Import Wizard opens a login dialog. This dialog collects the connection information for the SQLAnywhere database.

Fill in the following fields in the **Login/Connection** dialog:

Database

Select the SQLAnywhere database that contains the tables to import. If the **Database** list is empty, you may type in the name. See your server documentation or your DBA for information on database names.

Username

For Standard Security, type your MSSQL Username. For Trusted Security (Integrated NT Security) no Username is required. See your server documentation or your DBA for information on applicable Usernames and security methods.

Password

For Standard Security, type your MSSQL Password. For Trusted Security (Integrated NT Security) no Password is required. See your server documentation or your DBA for information on applicable Passwords and security methods.

Filter

Optionally, provide a filter expression to limit the list of tables and views to import. The filter expression queries the SYSCATALOG view. The filter expression is limited to 1024 characters in length.

Tip: The filter is case sensitive, so type your filter value accordingly.

Following is a list of the column names (and their Clarion datatypes) you can reference in your filter expression. See your SQLAnywhere documentation for information on these fields.

CREATOR	STRING(128)
TNAME	STRING(128)
DBSPACENAME	STRING(128)
TABLETYPE	STRING(10)
NCOLS	LONG
PRIMARY_KEY	STRING(1)
CHECK	STRING(32767)
REMARKS	STRING(32767)

Disconnect from Server when Import is finished

Check this box to disconnect from the server after importing (or canceling). Generally, you should clear this box when importing multiple definitions in order to maintain your connection to the server between imports.

Next >

Press this button to open the Import Wizard's **Import List** dialog.

SQLAnywhere Accelerator SQL Import Wizard--Import List Dialog

When you press the **Next >** button, the Import Wizard opens the **Import List** dialog. The **Import List** dialog lists the importable items.

Highlight the table whose definition to import, then press the **Finish** button to import. The Import Wizard adds the definition to your Clarion Data Dictionary, then opens the **File Properties** dialog to let you modify the default definition.

Import additional tables by repeating these steps. After all the items are imported, return to the Dictionary Editor where you can define relationships and delete any columns not used in your Clarion application. See *SQL Accelerators--Define Only the Fields You Use*.

SQLAnywhere Accelerator Connection Information

(and Driver Configuration--File Properties)

Typically, you add SQLAnywhere support to your application by importing the table definitions into your Clarion Data Dictionary. The Import Wizard automatically fills in the **File Properties** dialog with default values based on the imported item.

The OWNER attribute for SQLAnywhere Accelerator takes the format:

`database ,username ,password`

Tip: Type an exclamation point (!) followed by a variable name in the Owner Name field of the File Properties dialog to specify a variable connect string rather than hard coding the connect string (OWNER attribute) . For example: !GLO:ConnectionString.

SQLAnywhere Accelerator Driver Strings

There are switches or "driver strings" you can set to control the way your application creates, reads, and writes files with a specific driver. Driver strings are simply messages or parameters that are sent to the file driver at run-time to control its behavior. See *Common Driver Features--Driver Strings* for an overview of these runtime Database Driver switches and parameters.

Tip: A forward slash precedes all SQL Accelerator driver strings. The slash allows the driver to distinguish between driver strings and SQL statements sent with SEND.

In addition to the standard SQL Driver Strings, the SQLAnywhere Accelerator supports the following Driver Strings:

LOGONSCREEN (SQLAnywhere Accelerator)

```
DRIVER('SQLAnywhere', '/LOGONSCREEN = TRUE | FALSE ')
```

```
[ AutoLogon" = ] SEND(file, '/LOGONSCREEN [ = TRUE | FALSE ]')
```

See Also: PROP:LogonScreen.

GATHERATOPEN (SQLAnywhere Accelerator)

```
DRIVER('SQLAnywhere', '/GATHERATOPEN = TRUE | FALSE ')
```

By default the driver delays gathering field information until it is required. However, some backends (like Sybase 11) perform poorly under these conditions. Setting GATHERATOPEN to TRUE forces the driver to gather most of the field information when the file is opened, which avoids a slowdown during program execution.

SQLAnywhere Accelerator Driver Properties

You can use Clarion's property syntax to query and set certain SQLAnywhere Accelerator driver properties. In addition to the standard SQL Accelerator properties (see *SQL Accelerators--SQL Accelerator Properties*), the SQLAnywhere Accelerator supports the following properties.

PROP:LogonScreen (SQLAnywhere Accelerator)

PROP:LogonScreen sets or returns the toggle that determines whether the driver automatically prompts for logon information. By default (PROP:LogonScreen=True), the driver does display a logon window if no connect string is supplied. If set to False and there is no connect string, the OPEN(file) fails and FILEERRORCODE() returns '28000.' For example:

```
Afile FILE,DRIVER('SQLAnywhere')
!file declaration with no userid and password
    END
CODE
Afile{PROP:LogonScreen}=True    !enable auto login screen
OPEN(Afile)
```

The logon screen is the SQLAnywhere Connect dialog. Consult your SQLAnywhere documentation for more information on this dialog. The end-user's ability to use the connect dialog will depend on the security surrounding the SQLAnywhere database. For example, the end-users may have access rights to a named database (sademo) that they can access with the SQLAnywhere client software, but they may not have access rights to the *.db files that comprise the database. The SQLAnywhere connect dialog requires *.db files rather than database name.

Using Embedded SQL(SQLAnywhere Accelerator)

You can use Clarion's property syntax (PROP:SQL) to send SQL statements to the backend SQL server within the normal execution of your program. See *SQL Accelerators--Using Embedded SQL* for more information.

Calling a Stored Procedure (SQLAnywhere Accelerator)

For SQLAnywhere NORESULTCALL is more efficient than CALL.

SQLAnywhere Accelerator Supported File Commands and Functions

File Attributes	Supported
CREATE	Y
DRIVER(<i>filetype</i> [, <i>driver string</i>])	Y
NAME	Y
ENCRYPT	N
OWNER(<i>password</i>)	Y ₁
RECLAIM	N
PRE(<i>prefix</i>)	Y
BINDABLE	Y
THREAD	Y
EXTERNAL(<i>member</i>)	Y
DLL(<i>flag</i>)	Y
OEM	N
File Structures	Supported
INDEX	Y
KEY	Y
MEMO	N
BLOB	Y
RECORD	Y
Index, Key, Memo Attributes	Supported
BINARY	N ₃
DUP	Y
NOCASE	Y
OPT	N
PRIMARY	Y

NAME		Y
Ascending Components		Y
Descending Components		Y
Mixed Components		Y
Field Attributes		Supported
DIM		N
OVER		Y
NAME		Y
File Procedures		Supported
BOF(<i>file</i>)	N	
BUFFER(<i>file</i>)		Y
BUILD(<i>file</i>)		Y
BUILD(<i>key</i>)		Y
BUILD(<i>index</i>)		Y ₃
BUILD(<i>index, components</i>)		Y ₃
BUILD(<i>index, components, filter</i>)		N
BYTES(<i>file</i>)		Y
CLOSE(<i>file</i>)		Y
COPY(<i>file, new file</i>)		N
CREATE(<i>file</i>)		Y
DUPLICATE(<i>file</i>)		Y
DUPLICATE(<i>key</i>)		Y
EMPTY(<i>file</i>)		Y
EOF(<i>file</i>)	N	
FLUSH(<i>file</i>)		N
LOCK(<i>file</i>)		N
NAME(<i>label</i>)		Y

OPEN(<i>file, access mode</i>)	Y
PACK(<i>file</i>)	N
POINTER(<i>file</i>)	N
POINTER(<i>key</i>)	N
POSITION(<i>file</i>)	N
POSITION(<i>key</i>)	Y
RECORDS(<i>file</i>)	Y
RECORDS(<i>key</i>)	Y
REMOVE(<i>file</i>)	Y
RENAME(<i>file, new file</i>)	N
SEND(<i>file, message</i>)	Y
SHARE(<i>file, access mode</i>)	Y
STATUS(<i>file</i>)	Y
STREAM(<i>file</i>)	N
UNLOCK(<i>file</i>)	N
Record Access	Supported
ADD(<i>file</i>)	Y
ADD(<i>file, length</i>)	N
APPEND(<i>file</i>)	Y
APPEND(<i>file, length</i>)	N
DELETE(<i>file</i>)	Y
GET(<i>file, key</i>)	Y
GET(<i>file, filepointer</i>)	N
GET(<i>file, filepointer, length</i>)	N
GET(<i>key, keypointer</i>)	N
HOLD(<i>file</i>)	N
NEXT(<i>file</i>)	Y

NOMEMO(<i>file</i>)	N
PREVIOUS(<i>file</i>)	Y
PUT(<i>file</i>)	Y
PUT(<i>file</i> , <i>filepointer</i>)	N
PUT(<i>file</i> , <i>filepointer</i> , <i>length</i>)	N
RELEASE(<i>file</i>)	N
REGET(<i>file</i> , <i>string</i>)	N
REGET(<i>key</i> , <i>string</i>)	Y
RESET(<i>file</i> , <i>string</i>)	N
RESET(<i>key</i> , <i>string</i>)	Y
SET(<i>file</i>)	Y
SET(<i>file</i> , <i>key</i>)	N
SET(<i>file</i> , <i>filepointer</i>)	N
SET(<i>key</i>)	Y
SET(<i>key</i> , <i>key</i>)	Y
SET(<i>key</i> , <i>keypointer</i>)	N
SET(<i>key</i> , <i>key</i> , <i>filepointer</i>)	N
SKIP(<i>file</i> , <i>count</i>)	Y
WATCH(<i>file</i>)	Y
Transaction Processing	Supported (see Note 2)
LOGOUT(<i>timeout</i> , <i>file</i> , ..., <i>file</i>)	Y ⁴
COMMIT	Y
ROLLBACK	Y
Null Data Processing	Supported
NULL(<i>field</i>)	Y
SETNULL(<i>field</i>)	Y
SETNONNULL(<i>field</i>)	Y

Notes:

- 1 We recommend using a variable password that is lengthy and contains special characters because this more effectively hides the password value from anyone looking for it. For example, a password like "dd...#\$...*&" is much more difficult to "find" than a password like "SALARY."

Tip: To specify a variable instead of the actual password in the Owner Name field of the File Properties dialog, type an exclamation point (!) followed by the variable name. For example: !MyPassword.

- 2 See also *PROP:Logout* in the *Language Reference*.
- 3 BUILD(index) sets internal driver flags to guarantee the driver generates the correct ORDER BY clause. The driver does not call the backend server.
- 4 Whether LOGOUT also LOCKs the table depends on the server's configuration for transaction processing. See your server documentation.

SQLAnywhere Accelerator Synchronizer Server

Clarion's Enterprise Edition includes the SQLAnywhere Synchronizer Server and the Data Dictionary Synchronizer. The Dictionary Synchronizer uses the Synchronizer Server to gather complete information about an SQLAnywhere database.

The SQLAnywhere Synchronizer Server is one of several used by the Dictionary Synchronizer. All the common behavior of all the SQL Accelerators is documented in the *SQL accelerators* section. All behavior specific to this driver is noted here.

SQLAnywhere Accelerator Synchronizer Login Dialog

Clarion's Dictionary Synchronizer Wizard (Enterprise Edition) lets you import an entire SQLAnywhere database definition into your Clarion Data Dictionary in a single pass. During this process, the Synchronizer Wizard opens an SQLAnywhere login dialog. This dialog collects the connection information for the SQLAnywhere database.

Fill in the following fields in the login dialog:

Database

Select the SQLAnywhere database that contains the tables or views to import. If the **Database** list is empty, you may type in the name. See your server documentation or your DBA for information on database names.

Username

Type your SQLAnywhere Username. See your server documentation or your DBA for information on Usernames.

Password

Type your SQLAnywhere Password. See your server documentation or your DBA for information on Passwords.

Include System Files

Select this option to include system tables in the list of importable objects.

Exclude System Files

Select this option to exclude system tables from the list of importable objects.

Other Filter

Select this option to provide a filter expression to limit the list of tables and views to import. The filter expression queries the SYSCATALOG view. The filter expression is limited to 1024 characters in length.

Tip: The filter is case sensitive, so type your filter value accordingly.

Following is a list of the column names (and their Clarion datatypes) you can reference in your filter expression. See your SQLAnywhere documentation for information on these fields.

CREATOR	STRING(128)
TNAME	STRING(128)
DBSPACENAME	STRING(128)
TABLETYPE	STRING(10)
NCOLS	LONG
PRIMARY_KEY	STRING(1)
CHECK	STRING(32767)
REMARKS	STRING(32767)

Index:

- 32-bit applications.....258
- Alias195
- ALL_CATALOG263
- ALLOWDETAILS182
- AlwaysRebind.....195
- APPENDBUFFER182
- auto numbered keys267
- BINDCOLORDER.....182
- Btrieve
 - Driver Properties38
- C60ORA.DLL.....261
- Calling a Stored Procedure
 - SQLAnywher308
- Calling a Stored Procedure (SQLAnywhere Accelerator).....308
- Calling a xe "Stored Procedures Oracle"Stored Procedure
 - Oracle Accelerator.....276
- CHECKFORNULL180
- Choosing the Right Database Driver.....6
- Clipper
 - Driver Strings62
 - Other71
- Common Database Driver Features.....7
- ConnectionString195
- Database Drivers.....5
- Database Name
 - Scalable SQL.....214, 313
 - Scalable SQL.....228
 - SQLAnywhere301
- dBaselll
 - Driver Strings78
 - Other87
- dBaselV
 - Driver Strings94
 - Other104
- DBMSver196
- Debugging Your ODBC Application248
- Debugging Your SQL Application.....173
- Details.....196
- Disconnect.....196
- DLL Coding Practices.....276
- DOS
 - Driver Strings111
- Driver Properties
 - MSSQL.....228
 - SQLAnywhere307
- Driver Strings10
 - MSSQL.....224
 - Oracle.....269
 - SQLAnywhere305
- Embedded SQL204
 - Oracle.....274
- FILTER.....214, 215, 314
- FoxPro
 - Driver Strings120
 - Other129
- Future Oracle Releases.....257
- GATHERATOPEN184, 226, 306
- GATHERATOPEN (MSSQL Accelerator)226
- General Information for all SQL Drivers ..153
- GETINFO184
- hdbc198
- henv198
- HINT.....225, 270
 - Oracle.....270
- hstmt198
- Import Wizard
 - Oracle.....262, 263
 - Scalable SQL214
 - SQLAnywhere301
- Importing from ODBC Data Sources245
- Inner198
- IPX host262, 266
- ISOLATIONLEVEL186
- JOINTYPE187
- Key Properties dialog.....267
- local host.....266
- LogFile201
- LOGFILE.....190
- LoginTimeout201
- logon
 - MSSQL.....228
 - SQLAnywhere307
- LOGON SCREEN271
- LogonScreen200
- LOGONSCREEN
 - MSSQL.....226

LOGONSCREEN (MSSQL Accelerator) .226	Oracle Accelerator Driver Properties274
LOGONSCREEN (SQLAnywhere Accelerator).....306	Oracle Accelerator Driver Strings269
MSSQL	Oracle Accelerator Generating Unique Key Values267
Driver Properties228	Oracle Accelerator Installation.....260
Driver Strings224	Oracle Accelerator Performance Considerations265
LOGONSCREEN226	Oracle Accelerator Supported Commands and Attributes277
SAVESTOREDPROC226	Oracle Accelerator System Requirements258
TRUSTEDCONNECTION.....227	Oracle Accelerator Table Import Wizard—Import List Dialog265
MSSQL Accelerator Calling a Stored Procedure218	Oracle Accelerator Table Import Wizard—Login Dialog262
MSSQL Accelerator Connection Information and Driver Configuration--File Properties217	Oracle Accelerator Troubleshooting285
MSSQL Accelerator Driver Properties.....228	Oracle Accelerator --Using Embedded SQL274
MSSQL Accelerator Driver Strings224	Oracle connection.....262
MSSQL Accelerator Overview213	Oracle CONSTRAINTs262
MSSQL Accelerator Performance Considerations218	Oracle Driver Strings269
MSSQL Accelerator SQL Import Wizard--Import List Dialog216	Oracle hints270
MSSQL Accelerator SQL Import Wizard--Login Dialog214	Oracle indexes262
MSSQL Accelerator Supported File Commands and Functions230	Oracle linking fields.....273
MSSQL Accelerator Synchronizer Server235	Oracle login.....262
MSSQL Accelerator Using Embedded SQL221	Oracle Login Dialog266
MSSQL logon228	Oracle Personal271
NORESULTCALL	Oracle Sequences267
Oracle.....276	Oracle version.....258, 259
ODBC Connection Information and Driver Configuration--File Properties246	Oracle versions257
ODBC Data Types244	OrderAllTables201
ODBC Driver Properties255	OrderInSelect.....202
ODBC Pros and Cons240	Password
ODBC Supported Commands and Attributes249	SQLAnywhere301
Oracle	PASSWORD266
auto numbered keys267	PATH
sequence numbers267	Oracle.....258, 259
Unique Key Values267	Performance Considerations159
Oracle Accelerator257	PERSONAL271
Oracle Accelerator Automatic Login Dialog266	Oracle.....271
Oracle Accelerator Data Types283	PERSONAL Oracle.....271
	PL/SQL275
	Profile.....202
	Prop
	Name.....169
	PROP225
	Alias.....195

- AlwaysRebind 195
- ConnectionString 195
- DBMSver 196
- Details 196
- Disconnect 196
- GroupBy 197
- Having 171, 172, 197
- hdbc 198
- henv 198
- Hint 198, 225
 - Oracle 198
- hstmt 198
- Inner 198
- IsolationLevel 199
- Log 200
- LogFile 201
- LoginTimeout 201
- LogonScreen 200
 - MSSQL 228, 229
 - SQLAnywhere 307
- LogonScreen (MSSQL Accelerator) 228
- LogonScreen (SQLAnywhere Accelerator) 307
- OrderAllTables 201
- OrderInSelect 202
- Profile 202
- QuoteString 202
- SQL 203
- SQLFilter 205
- SQLJoinExpression 206
- SQLOrder 207
- protocol
 - local
 - IPX
 - TCP/IP 266
- QuoteString 202
- readme 260
- Registering the Oracle Accelerator 261
- Registering the Oracle Driver in Clarion.. 260
- SAVESTOREDPROC
 - MSSQL 226
- SAVESTOREDPROC (MSSQL Accelerator) 226
- Scalable SQL Supported Commands and Attributes 293
- Setup 260
- SQL 203, 204
- SQL Accelerator Drivers
 - Supported Commands and Attributes .. 175
- SQL Driver Behavior 157
- SQL Driver Strings 181
- SQL statements
 - Oracle 257
 - Scalable SQL 213, 299
- SQL Views 169
- SQLAnywhere
 - Driver Properties 307
 - Driver Strings 305
- SQLAnywhere Accelerator Synchronizer Server 313
- SQLAnywhere Accelerator Connection Information and Driver Configuration--File Properties 304
- SQLAnywhere Accelerator Driver Properties 307
- SQLAnywhere Accelerator Driver Strings 305
- SQLAnywhere Accelerator Overview 299
- SQLAnywhere Accelerator SQL Import Wizard--Import List Dialog 303
- SQLAnywhere Accelerator SQL Import Wizard--Login Dialog 301
- SQLAnywhere Accelerator Supported File Commands and Functions 309
- SQLAnywhere Accelerator Synchronizer Login Dialog 313
- SQLAnywhere logon 307
- SQLFilter 205
- SQLJoinExpression 206
- SQLOrder 207
- Start the SQLAnywhere Client 300
- Stored procedures 167
- TCP/IP host 262, 266
- TopSpeed Database Recovery Utility 149
- TPSFIX Command Line Parameters 151
- TRUSTEDCONNECTION
 - MSSQL 227
- TRUSTEDCONNECTION (MSSQL Accelerator) 227
- USEASYNCHRONOUSCALLS 272
- USEINNERJOIN 191
- Username 262, 263, 266
 - SQLAnywhere 301
- Using Embedded SQL 164

Using Embedded SQL(SQLAnywhere Accelerator).....	308	Version.....	196
Using SQL Tables in your Clarion Application	154	Views - SQL.....	169
Variable file names		WHERE.....	193
Scalable SQL.....	217, 230, 304, 309	Oracle.....	273
VERIFYVIASELECT	192	ZERODATE	194
		ZEROISNULL	194