Clarion

Advanced Topics and Reference Guide

SoftVelocity Incorporated

2335 East Atlantic Blvd. Suite 410

Pompano Beach, Florida 33062

(954) 785-4555

www.softvelocity.com

Documentation for Clarion 7

Printed in the United States of America (0409)

Table Of Contents

Advanced Topics

General Topics

This section presents miscellaneous topics that are geared to the more advanced developer, and provides a potpourri of information:

Clarion 7 Migration Tips

This topic is designed as a quick reference for developers and programmers who are migrating applications to Clarion 7 from prior versions.

The following are changes in that were originally documented in the Clarion 6 versions that need to be reviewed by all developers, particularly those migrating applications *prior* to Clarion 6 directly to the Clarion 7 environment:

Change of EVALUATE Error Codes

The error codes posted by the EVALUATE statement have been modified in Clarion 6:

1010 - formerly 800: Bad expression syntax

1011 - formerly 801: Unknown identifier in the expression

1012 - formerly 802: Mismatched POPBIND

Default Splash Procedure Icon

Loading a C55 application into Clarion 6 that uses the default splash screen will produce a link error that says 'C55.GIF' not found.

The best workaround is to simply change the icon, or just delete the old splash procedure and add a new one using the Clarion 6 default.

DLL Initialization

Enforcement of threaded variables in multi-DLL applications is critical in Clarion 6. In older versions, if your file definitions are set to "open in current thread" in the dictionary (the THREAD attribute is set in the FILE definition), your file definitions in your DLLs must match that definition. To ensure this, examine each application's global file control section, and make sure that all of your files are set to 'ALL THREADED' in the **Threaded** drop list.

Regarding files that are defined across multiple DLLs (e.g., a Customer file defined in a data DLL and again in another DLL), ***you CANNOT mix thread and non-threaded attributes on the same files in a multi-dll application***. Although this programming style was permitted in earlier versions of Clarion, the initialization of preemptive threads will not allow this in Clarion 6.

You can use *either* setting, thread or non-threaded, as long as it's consistent across all DLLs and your executable. And, of course, mixing the threaded and non-threaded attributes among different files is still permitted.

Embedding code when closing a Process procedure

The Process procedure used with the ABC templates calls ThisProcess.Close() after ThisWindow.Kill has fully completed. Consequently, any object created in the scope of the process window which is called inside ThisProcess.Close() will cause a GPF since the destructor for that object will already have completed during ThisWindow.Kill.

In Clarion 6, the ViewManager class destructor used with the ProcessClass is now calling ThisProcess.Close() to make sure that the VIEW is closed. This was not needed in previous versions of Clarion because local VIEWs were automatically closed when a procedure exited. With the new threading model, local VIEWs are now not automatically closed until the thread is destroyed.

There is a distinct chance that any call to a local object inside ThisWindow.Close() will cause a GPF when exiting the process procedure, because it has already been disposed by the time the final ThisProcess.Close() call happens.

Anyone embedding source code in the ThisWindow.Close() method needs to add some kind of condition surrounding any call to a local object that stops it happening after ThisWindow.Kill() has occurred.

General Rules regarding your data and the new Thread Model

1. Use the THREAD attribute on global Data (file, class, group, queue or simple types).
2. Use the THREAD attribute on module Data (file, class, group, queue or simple types).
3. Avoid the use of static variables.
4. Don't pass the address of anything within a START command - this was a common trick used by people to communicate between threads.

If you do any of the above you must make them thread safe. Refer to the *Multi Threaded Programming* PDF for detailed information on this process.

Local data (including classes that are normally instantiated locally) is automatically threaded (unless you put the STATIC attribute on it).

Global EIP Classes

Some applications that use edit-in-place in prior versions will migrate to Clarion 6 with the *EditClass* defined as the default edit-in-place class. In Clarion 6, this class needs to be changed to the *EditEntryClass*. To do this, access the *Global Properties* in the Application Generator. In the **Classes** tab, press the Browser button and change the default edit in place class from *EditClass* to *EditEntryClass*.

Global Pools and GROUPs

In Clarion versions prior to 5.5H, if a GROUP were defined in the dictionary's Global Pool without using a prefix, the prefix of the Global Pool (which is required) would be used in its place.

For example:

```
GLOB:GROUPONE GROUP,PRE() !`GLOB' is the prefix of the Global Pool
GLOB:STRING1   STRING(20)
GLOB:STRING2   STRING(20)

END
```

With Clarion 6, this same group will be generated without the Global prefix, and would appear as follows:

```
GLOB:GROUPONE GROUP,PRE() !`GLOB' is the prefix of the Global Pool
STRING1   STRING(20)
STRING2   STRING(20)

END
```

Since it is possible to define multiple Global Pools in your dictionary, a second Global Pool could be created as follows:

```
GLO2:GROUPTWO GROUP,PRE() !`GLO2' is the prefix of the 2nd Global Pool
GLO2:STRING1   STRING(20)
GLO2:STRING2   STRING(20)

END
```

Which in Clarion 6 will be generated as follows:

```
GLO2:GROUPTWO GROUP,PRE() !`GLO2' is the prefix of the 2nd Global Pool
STRING1   STRING(20)
STRING2   STRING(20)

END
```

This will cause the compiler to generate "Duplicate Label" warnings when migrating your applications. The solution to this is to add a prefix to each of your GROUPs that share common label names across Global Pools.

For example:

```
GLOB:GROUPONE GROUP,PRE(GLOB) !`GLOB' is the prefix of the 1st GROUP
STRING1   STRING(20)
STRING2   STRING(20)

END


GLO2:GROUPTWO GROUP,PRE(GLO2) !`GLO2' is the prefix of the 2nd GROUP
STRING1   STRING(20)
STRING2   STRING(20)

END
```

Heap Overflow Error when migrating applications

During the early testing phase of Clarion 6, it was noted that some applications would post a "Heap Overflow" error when attempting to load applications of prior Clarion versions into the Clarion 6 IDE.

In nearly all cases, the solution is to first export the application to a text file (TXA), and then import it as text into the Clarion 6 environment.

ISAM File Access Performance

Some users have reported that if there is any experience of slow file accesses when using ISAM files, switching off the **Defer opening files until accessed** in the application's *Global Properties* - **File Control** will improve the performance.

Large WINDOW structures

In each control that is populated in a WINDOW structure in Clarion 6, there is now extra information for each control that takes about 10 extra bytes per control. This may cause some large and complex windows to not import properly from prior versions.

You may need to shorten some use variables or remove controls and create/destroy them at runtime or redesign the window to make it a bit more efficient.

Migrating Large Dictionaries and Data Paths

One of the nice new features of this release is a new system property:
SYSTEM{PROP:DataPath}.

With this, you can set your data file names in the dictionary to have no path in them, and then set the data path once in the program start up. From there, each file will inherit the common data path.

With that in mind, dictionaries created in a prior version will continue to work. The only issue is where file names and structures are stored exclusively in a DLL and referenced from the EXE. In prior versions, you had to define these objects in the EXE as EXTERNAL, and did not care if the files in this object were threaded or not. In this release, any objects that contain threaded data must add the THREAD attribute to the object definition.

Mixing threaded and non-threaded data in an object is dangerous and likely to cause problems.

Migration of hand coded project files

All projects compiled in Clarion 6 are 32-bit. Prior to loading older project files (.PR or .PRJ) into the Clarion 6 environment, load them into your favorite text editor and make sure that the following pragma entry is set properly:

```
#system win32
```

You can also load the project into the project editor. When you press OK to close the project, the #system pragma will be automatically updated.

POINTER(File) and POSITION(File)

The behavior of POINTER(File) is different for different file systems. For example, the first record in a TopSpeed file doesn't have a pointer value of 1.

It may still be safe to use for certain file systems, but for code portability, POSITION(File) is the way to go.

Remove MDI attribute from dockable toolbar windows

In Clarion 6, the MDI attribute is no longer permitted on any toolbar window that is dockable (windows with the DOCK and TOOLBOX attributes).

RTF Text Control and RTF Control Templates

To all developers that used the RTF Control template in prior versions of Clarion, this template has been deprecated, and is now replaced by the new RTF Text Control template.

The RTF Control template is still included for users who wish to migrate their applications to Clarion 6, and avoid unnecessary template warnings or errors. Upon migration to Clarion 6, you will notice a message in the **Actions** tab of the control's properties that notifies you that this template should be replaced. You will also see a warning at source generation time that notifies you that this template has been deprecated.

TXA Comparison Technique

If you are having troubles with applications converted to Clarion 6 using DLLs, there is a possibility that the DLL that was converted contained hidden information (like a third party library) that was not detected by the conversion process.

To confirm this, try the following:

Export the old DLL application to TXA format (Export Text)

Export a new Clarion 6 DLL application to TXA format (Export Text)

Next, compare the TXA's up through the first procedure (i.e., the program/global area). This might give you some ideas regarding information converted from an old application that may not be compatible, or does not exist in Clarion 6.

Use of Error Managers during DLL Initialization

A change has also been made in the DLL initialization of ABC-based applications. During initialization, the DLL uses a LocalErrors Class rather than the Global executable's GlobalErrors Class.

For example, in a multi DLL application and during initialization of the DLL containing Global data, if errors need to be posted to the error manager, they will be posted to the DLL's local error manager (LocalErrors) instead of the application's global error manager. The reason for this is that the DLL's error manager is not set to use the application's error manager until *after* initialization of the DLL. *During* initialization, the DLL uses the LocalErrors Manager rather than the executable's GlobalErrors Manager. Inside the DLL *Init* procedure, extra code is generated to assign GlobalErrors, and also assign the passed error manager to the already initialized file managers and relation managers.

Developers who modified the global error manager in their applications using DLLs will now need to be aware of the new local error managers that are applied.

Use of ABC Classes with Clarion template based applications

This release of Clarion includes many new template features that are now supported in the Clarion template chain, but rely on the use of the ABC Library Classes. Examples of this are the support for RTF Text Controls, Pop-up Calendars, and the new enhanced edit-in-place.

The linking of the ABC class support requires the following default pragmas settings to be included in the project system:

```
_ABCDllMode_=>0
_ABCLinkMode_=>1
```

Template support to automatically include these pragmas can be found in the *Global Properties* **Classes** tab by checking the **Enable the use of ABC Classes** check box.

Commonly Used Equates

The following topic displays the common EQUATES used by the Clarion IDE, as listed in the EQUATES.CLW and TPLEQU.CLW files.

Template Equates

Event numbers - Field-specific

Event numbers - Field-independent

Standard Window Equates

```
! Event numbers

 Field-specific events (FIELD() returns field number)

EVENT:Accepted      EQUATE (01H)
EVENT:NewSelection  EQUATE (02H)
EVENT:ScrollUp      EQUATE (03H)
EVENT:ScrollDown    EQUATE (04H)
EVENT:PageUp        EQUATE (05H)
EVENT:PageDown      EQUATE (06H)
EVENT:ScrollTop     EQUATE (07H)
EVENT:ScrollBottom  EQUATE (08H)
EVENT:Locate        EQUATE (09H)

EVENT:MouseDown     EQUATE (01H)
EVENT:MouseUp       EQUATE (0aH)
EVENT:MouseIn       EQUATE (0bH)
EVENT:MouseOut      EQUATE (0cH)
EVENT:MouseMove     EQUATE (0dH)
EVENT:AlertKey      EQUATE (0fH)
EVENT:PreAlertKey   EQUATE (10H)
EVENT:Dragging      EQUATE (11H)
EVENT:Drag          EQUATE (12H)
EVENT:Drop          EQUATE (13H)
EVENT:ScrollDrag    EQUATE (14H)
EVENT:TabChanging   EQUATE (15H)
EVENT:Expanding     EQUATE (16H)
EVENT:Contracting   EQUATE (17H)
EVENT:Expanded      EQUATE (18H)
```

```
EVENT:Contracted    EQUATE (19H)
EVENT:Rejected      EQUATE (1AH)
EVENT:DroppingDown  EQUATE (1BH)
EVENT:DroppedDown   EQUATE (1CH)
EVENT:ScrollTrack   EQUATE (1DH)
EVENT:ColumnResize  EQUATE (1EH)
EVENT:Selecting     EQUATE (1FH)

EVENT:Selected      EQUATE (101H)

 Field-independent events (FIELD() returns 0)

EVENT:CloseWindow   EQUATE (201H)
EVENT:CloseDown     EQUATE (202H)
EVENT:OpenWindow    EQUATE (203H)
EVENT:OpenFailed    EQUATE (204H)
EVENT:LoseFocus     EQUATE (205H)
EVENT:GainFocus     EQUATE (206H)


EVENT:Timer         EQUATE (20BH)
EVENT:DDErequest    EQUATE (20CH)
EVENT:DDEadvise     EQUATE (20DH)
EVENT:DDEdata       EQUATE (20EH)
EVENT:DDEcommand    EQUATE (20FH)    ! same as DDEexecute
EVENT:DDEexecute    EQUATE (20FH)
EVENT:DDEpoke       EQUATE (210H)
EVENT:DDEclosed     EQUATE (211H)

EVENT:Move          EQUATE (220H)
EVENT:Size          EQUATE (221H)
EVENT:Restore       EQUATE (222H)
EVENT:Maximize      EQUATE (223H)
EVENT:Iconize       EQUATE (224H)
EVENT:Completed     EQUATE (225H)
EVENT:Moved         EQUATE (230H)
EVENT:Sized         EQUATE (231H)
EVENT:Restored      EQUATE (232H)
EVENT:Maximized     EQUATE (233H)
EVENT:Iconized      EQUATE (234H)
EVENT:Docked        EQUATE (235H)
```

```
EVENT:Undocked      EQUATE (236H)


EVENT:BuildFile     EQUATE (240H)
EVENT:BuildKey      EQUATE (241H)
EVENT:BuildDone     EQUATE (242H)


! User-definable events


EVENT:User          EQUATE (400H)
EVENT:Last          EQUATE (0FFFH)
STD:WindowList    EQUATE (1)
STD:TileWindow    EQUATE (2)
STD:CascadeWindow EQUATE (3)
STD:ArrangeIcons  EQUATE (4)
STD:HelpIndex     EQUATE (5)
STD:HelpOnHelp    EQUATE (6)
STD:HelpSearch    EQUATE (7)
STD:Help          EQUATE (8)
STD:Cut           EQUATE (10)
STD:Copy          EQUATE (11)
STD:Paste         EQUATE (12)
STD:Clear         EQUATE (13)
STD:Undo          EQUATE (14)
STD:Close         EQUATE (15)
STD:PrintSetup    EQUATE (16)
STD:TileHorizontal EQUATE (17)
STD:TileVertical   EQUATE (18)
CURSOR:None       EQUATE ('<0FFH,01H,00H,00H>')
CURSOR:Arrow      EQUATE ('<0FFH,01H,01H,7FH>')
CURSOR:IBeam      EQUATE ('<0FFH,01H,02H,7FH>')
CURSOR:Wait       EQUATE ('<0FFH,01H,03H,7FH>')
CURSOR:Cross      EQUATE ('<0FFH,01H,04H,7FH>')
CURSOR:UpArrow    EQUATE ('<0FFH,01H,05H,7FH>')
CURSOR:Size       EQUATE ('<0FFH,01H,81H,7FH>')
CURSOR:Icon       EQUATE ('<0FFH,01H,82H,7FH>')
CURSOR:SizeNWSE   EQUATE ('<0FFH,01H,83H,7FH>')
CURSOR:SizeNESW   EQUATE ('<0FFH,01H,84H,7FH>')
CURSOR:SizeWE     EQUATE ('<0FFH,01H,85H,7FH>')
CURSOR:SizeNS     EQUATE ('<0FFH,01H,86H,7FH>')
CURSOR:DragWE     EQUATE ('<0FFH,02H,01H,7FH>')
CURSOR:Drop       EQUATE ('<0FFH,02H,02H,7FH>')
```

```
CURSOR:NoDrop      EQUATE ('<0FFH,02H,03H,7FH>')
CURSOR:Zoom        EQUATE ('<0FFH,02H,04H,7FH>')
ICON:None          EQUATE ('<0FFH,01H,00H,00H>')
ICON:Application   EQUATE ('<0FFH,01H,01H,7FH>')
ICON:Hand          EQUATE ('<0FFH,01H,02H,7FH>')
ICON:Question      EQUATE ('<0FFH,01H,03H,7FH>')
ICON:Exclamation   EQUATE ('<0FFH,01H,04H,7FH>')
ICON:Asterisk      EQUATE ('<0FFH,01H,05H,7FH>')
ICON:Pick          EQUATE ('<0FFH,02H,01H,7FH>')
ICON:Save          EQUATE ('<0FFH,02H,02H,7FH>')
ICON:Print         EQUATE ('<0FFH,02H,03H,7FH>')
ICON:Paste         EQUATE ('<0FFH,02H,04H,7FH>')
ICON:Open          EQUATE ('<0FFH,02H,05H,7FH>')
ICON:New           EQUATE ('<0FFH,02H,06H,7FH>')
ICON:Help          EQUATE ('<0FFH,02H,07H,7FH>')
ICON:Cut           EQUATE ('<0FFH,02H,08H,7FH>')
ICON:Copy          EQUATE ('<0FFH,02H,09H,7FH>')
ICON:Child         EQUATE ('<0FFH,02H,0AH,7FH>')
ICON:Frame         EQUATE ('<0FFH,02H,0BH,7FH>')
ICON:Clarion       EQUATE ('<0FFH,02H,0CH,7FH>')
ICON:NoPrint       EQUATE ('<0FFH,02H,0DH,7FH>')
ICON:Zoom          EQUATE ('<0FFH,02H,0EH,7FH>')
ICON:NextPage      EQUATE ('<0FFH,02H,0FH,7FH>')
ICON:PrevPage      EQUATE ('<0FFH,02H,10H,7FH>')
ICON:JumpPage      EQUATE ('<0FFH,02H,11H,7FH>')
ICON:Thumbnail     EQUATE ('<0FFH,02H,12H,7FH>')
ICON:Tick          EQUATE ('<0FFH,02H,13H,7FH>')
ICON:Cross         EQUATE ('<0FFH,02H,14H,7FH>')
ICON:Connect       EQUATE ('<0FFH,02H,15H,7FH>')
ICON:Print1        EQUATE ('<0FFH,02H,16H,7FH>')
ICON:Ellipsis      EQUATE ('<0FFH,02H,17H,7FH>')

ICON:VCRtop          EQUATE ('<0FFH,02H,81H,7FH>')
ICON:VCRrewind       EQUATE ('<0FFH,02H,82H,7FH>')
ICON:VCRback         EQUATE ('<0FFH,02H,83H,7FH>')
ICON:VCRplay         EQUATE ('<0FFH,02H,84H,7FH>')
ICON:VCRfastforward EQUATE ('<0FFH,02H,85H,7FH>')
ICON:VCRbottom       EQUATE ('<0FFH,02H,86H,7FH>')
ICON:VCRlocate       EQUATE ('<0FFH,02H,87H,7FH>')

BEEP:SystemDefault       EQUATE (0000H)
```

```
BEEP:SystemHand          EQUATE (0010H)
BEEP:SystemQuestion      EQUATE (0020H)
BEEP:SystemExclamation   EQUATE (0030H)
BEEP:SystemAsterisk      EQUATE (0040H)
REJECT:RangeHigh         EQUATE(1)  ! Above top range on SPIN
REJECT:RangeLow          EQUATE(2)  ! below bottom range ditto
REJECT:Range             EQUATE(3)  ! Other range error
REJECT:Invalid           EQUATE(4)  ! Invalid input


INCLUDE 'property.clw'
COLOR:NONE               EQUATE (-1)
COLOR:SCROLLBAR          EQUATE (80000000H)
COLOR:BACKGROUND         EQUATE (80000001H)
COLOR:ACTIVECAPTION      EQUATE (80000002H)
COLOR:INACTIVECAPTION    EQUATE (80000003H)
COLOR:MENU               EQUATE (80000004H)
COLOR:WINDOW             EQUATE (80000005H)
COLOR:WINDOWFRAME        EQUATE (80000006H)
COLOR:MENUTEXT           EQUATE (80000007H)
COLOR:WINDOWTEXT         EQUATE (80000008H)
COLOR:CAPTIONTEXT        EQUATE (80000009H)
COLOR:ACTIVEBORDER       EQUATE (8000000AH)
COLOR:INACTIVEBORDER     EQUATE (8000000BH)
COLOR:APPWORKSPACE       EQUATE (8000000CH)
COLOR:HIGHLIGHT          EQUATE (8000000DH)
COLOR:HIGHLIGHTTEXT      EQUATE (8000000EH)
COLOR:BTNFACE            EQUATE (8000000FH)
COLOR:BTNSHADOW          EQUATE (80000010H)
COLOR:GRAYTEXT           EQUATE (80000011H)
COLOR:BTNTEXT            EQUATE (80000012H)
COLOR:INACTIVECAPTIONTEXT EQUATE (80000013H)
COLOR:BTNHIGHLIGHT         EQUATE (80000014H)


COLOR:Black              EQUATE (0000000H)
COLOR:Maroon             EQUATE (0000080H)
COLOR:Green              EQUATE (0008000H)
COLOR:Olive              EQUATE (0008080H)
COLOR:Navy               EQUATE (0800000H)
COLOR:Purple             EQUATE (0800080H)
COLOR:Teal               EQUATE (0808000H)
COLOR:Gray               EQUATE (0808080H)
```

```
COLOR:Silver            EQUATE (0C0C0C0H)
COLOR:Red               EQUATE (00000FFH)
COLOR:Lime              EQUATE (000FF00H)
COLOR:Yellow            EQUATE (000FFFFH)
COLOR:Blue              EQUATE (0FF0000H)
COLOR:Fuschia           EQUATE (0FF00FFH)
COLOR:Aqua              EQUATE (0FFFF00H)
COLOR:White             EQUATE (0FFFFFFH)


! Parameter to CREATE / Return value from PROP:type
CREATE:sstring          EQUATE (1)
CREATE:string           EQUATE (2)
CREATE:image            EQUATE (3)
CREATE:region           EQUATE (4)
CREATE:line             EQUATE (5)
CREATE:box              EQUATE (6)
CREATE:ellipse          EQUATE (7)
CREATE:entry            EQUATE (8)
CREATE:button           EQUATE (9)
CREATE:prompt           EQUATE (10)
CREATE:option           EQUATE (11)
CREATE:check            EQUATE (12)
CREATE:group            EQUATE (13)
CREATE:list             EQUATE (14)
CREATE:combo            EQUATE (15)
CREATE:spin             EQUATE (16)
CREATE:text             EQUATE (17)
CREATE:custom           EQUATE (18)
CREATE:menu             EQUATE (19)
CREATE:item             EQUATE (20)
CREATE:radio            EQUATE (21)
CREATE:menubar          EQUATE (22)     ! return value only
CREATE:application      EQUATE (24)     ! return value only
CREATE:window           EQUATE (25)     ! return value only
CREATE:report           EQUATE (26)     ! return value only
CREATE:header           EQUATE (27)
CREATE:footer           EQUATE (28)
CREATE:break            EQUATE (29)
CREATE:form             EQUATE (30)
CREATE:detail           EQUATE (31)
CREATE:ole              EQUATE (32)
```

```
CREATE:droplist         EQUATE (33)
CREATE:dropcombo        EQUATE (34)
CREATE:progress         EQUATE (35)

CREATE:sheet            EQUATE (37)
CREATE:tab              EQUATE (38)
CREATE:panel            EQUATE (39)
CREATE:rtf              EQUATE (40)

CREATE:sublist          EQUATE (CREATE:list + 0100H)  ! list part of a DROP or COMBO

CREATE:toolbar          EQUATE (128)
FONT:thin               EQUATE (100)
FONT:regular            EQUATE (400)
FONT:bold               EQUATE (700)
FONT:weight             EQUATE (07FFH)
FONT:fixed              EQUATE (0800H)
FONT:italic             EQUATE (01000H)
FONT:underline          EQUATE (02000H)
FONT:strikeout          EQUATE (04000H)

FONT:Screen             EQUATE(0)
FONT:Printer            EQUATE(1)
FONT:Both               EQUATE(2)
FONT:TrueTypeOnly       EQUATE(4)
FONT:FixedPitchOnly     EQUATE(8)
CHARSET:ANSI            EQUATE (  0)
CHARSET:DEFAULT         EQUATE (  1)
CHARSET:SYMBOL          EQUATE (  2)
CHARSET:MAC             EQUATE ( 77)
CHARSET:SHIFTJIS        EQUATE (128)
CHARSET:HANGEUL         EQUATE (129)
CHARSET:JOHAB           EQUATE (130)
CHARSET:GB2312          EQUATE (134)
CHARSET:CHINESEBIG5     EQUATE (136)
CHARSET:GREEK           EQUATE (161)
CHARSET:TURKISH         EQUATE (162)
CHARSET:HEBREW          EQUATE (177)
CHARSET:ARABIC          EQUATE (178)
CHARSET:BALTIC          EQUATE (186)
CHARSET:CYRILLIC        EQUATE (204)
```

```
CHARSET:THAI            EQUATE (222)
CHARSET:EASTEUROPE      EQUATE (238)
CHARSET:OEM             EQUATE (255)
PEN:solid               EQUATE (0)
PEN:dash                EQUATE (1)
PEN:dot                 EQUATE (2)
PEN:dashdot             EQUATE (3)
PEN:dashdotdot          EQUATE (4)
PEN:null                EQUATE (5)
PEN:insideframe         EQUATE (6)

FALSE                   EQUATE (0)
TRUE                    EQUATE (1)
LISTZONE:field          EQUATE(0)
LISTZONE:right          EQUATE(1)
LISTZONE:header         EQUATE(2)
LISTZONE:expandbox      EQUATE(3)
LISTZONE:tree           EQUATE(4)
LISTZONE:icon           EQUATE(5)
LISTZONE:nowhere        EQUATE(6)

BUTTON:OK               EQUATE (01H)
BUTTON:YES              EQUATE (02H)
BUTTON:NO               EQUATE (04H)
BUTTON:ABORT            EQUATE (08H)
BUTTON:RETRY            EQUATE (10H)
BUTTON:IGNORE           EQUATE (20H)
BUTTON:CANCEL           EQUATE (40H)
BUTTON:HELP             EQUATE (80H)

!DDE link types

DDE:auto                EQUATE (0)
DDE:manual              EQUATE (-1)
DDE:remove              EQUATE (-2)

! Types
  OMIT('***',_WIDTH32_)
SIGNED                  EQUATE(SHORT)
UNSIGNED                EQUATE(USHORT)
_nopos                  EQUATE(08000H)
```

```
  ***
  COMPILE('***',_WIDTH32_)
SIGNED                 EQUATE(LONG)
UNSIGNED               EQUATE(LONG)
_nopos                 EQUATE(080000000H)
  ***
BOOL                   EQUATE(SIGNED)

!DIRECTORY equates & TYPEs

!Old 8.3 filename support

ff_:NORMAL             EQUATE(0)
ff_:READONLY           EQUATE(1)
ff_:HIDDEN             EQUATE(2)
ff_:SYSTEM             EQUATE(4)
ff_:DIRECTORY          EQUATE(10H)
ff_:ARCHIVE            EQUATE(20H)
ff_:LFN                EQUATE(80H)



ff_:queue     QUEUE,PRE(ff_),TYPE
name            string(13)
date            long
time            long
size            long
attrib          byte
              END

!full filename support

FILE:MaxFileName EQUATE(256)
FILE:MaxFilePath EQUATE(260)

FILE:Queue    QUEUE,PRE(FILE),TYPE
Name            STRING(FILE:MaxFileName)
ShortName       STRING(13)
Date            LONG
Time            LONG
Size            LONG
```

```
Attrib          BYTE
              END

oleQ          QUEUE,TYPE
name            CSTRING(64)
clsid           CSTRING(64)
progid          CSTRING(64)
              END

!FileDialog/FileDialogA equates

FILE:Save         EQUATE(1)
FILE:KeepDir      EQUATE(2)
FILE:NoError      EQUATE(4)
FILE:Multi        EQUATE(8)
FILE:LongName     EQUATE(10H)
FILE:Directory    EQUATE(20H)
FILE:CreatePrompt EQUATE(40H)
FILE:AddExtension EQUATE(80H)

OCX:default     EQUATE(0)
OCX:16bit       EQUATE(1)
OCX:32bit       EQUATE(2)
OCX:1632bit     EQUATE(3)
DOCK:Left       EQUATE(1)
DOCK:Top        EQUATE(2)
DOCK:Right      EQUATE(4)
DOCK:Bottom     EQUATE(8)
DOCK:Float      EQUATE(16)

DOCK:All        EQUATE(31)

!TopSpeed File Flags

TPSREADONLY     EQUATE(1)

!Match Flag Values
Match:Simple        EQUATE(0)
Match:Wild          EQUATE(1)
Match:Regular       EQUATE(2)
```

```
Match:Soundex      EQUATE(3)
Match:NoCase       EQUATE(10H)   ! May be added to Simple,Wild and Regular


PAPER:LETTER              EQUATE(1)       ! Letter 8 1/2 x 11 in
PAPER:LETTERSMALL         EQUATE(2)       ! Letter Small 8 1/2 x 11 in
PAPER:TABLOID             EQUATE(3)       ! Tabloid 11 x 17 in
PAPER:LEDGER              EQUATE(4)       ! Ledger 17 x 11 in
PAPER:LEGAL               EQUATE(5)       ! Legal 8 1/2 x 14 in
PAPER:STATEMENT           EQUATE(6)       ! Statement 5 1/2 x 8 1/2 in
PAPER:EXECUTIVE           EQUATE(7)       ! Executive 7 1/4 x 10 1/2 in
PAPER:A3                  EQUATE(8)       ! A3 297 x 420 mm
PAPER:A4                  EQUATE(9)       ! A4 210 x 297 mm
PAPER:A4SMALL             EQUATE(10)      ! A4 Small 210 x 297 mm
PAPER:A5                  EQUATE(11)      ! A5 148 x 210 mm
PAPER:B4                  EQUATE(12)      ! B4 250 x 354
PAPER:B5                  EQUATE(13)      ! B5 182 x 257 mm
PAPER:FOLIO               EQUATE(14)      ! Folio 8 1/2 x 13 in
PAPER:QUARTO              EQUATE(15)      ! Quarto 215 x 275 mm
PAPER:10X14               EQUATE(16)      ! 10x14 in
PAPER:11X17               EQUATE(17)      ! 11x17 in
PAPER:NOTE                EQUATE(18)      ! Note 8 1/2 x 11 in
PAPER:ENV_9               EQUATE(19)      ! Envelope #9 3 7/8 x 8 7/8
PAPER:ENV_10              EQUATE(20)      ! Envelope #10 4 1/8 x 9 1/2
PAPER:ENV_11              EQUATE(21)      ! Envelope #11 4 1/2 x 10 3/8
PAPER:ENV_12              EQUATE(22)      ! Envelope #12 4 \276 x 11
PAPER:ENV_14              EQUATE(23)      ! Envelope #14 5 x 11 1/2
PAPER:CSHEET              EQUATE(24)      ! C size sheet
PAPER:DSHEET              EQUATE(25)      ! D size sheet
PAPER:ESHEET              EQUATE(26)      ! E size sheet
PAPER:ENV_DL              EQUATE(27)      ! Envelope DL 110 x 220mm
PAPER:ENV_C5              EQUATE(28)      ! Envelope C5 162 x 229 mm
PAPER:ENV_C3              EQUATE(29)      ! Envelope C3  324 x 458 mm
PAPER:ENV_C4              EQUATE(30)      ! Envelope C4  229 x 324 mm
PAPER:ENV_C6              EQUATE(31)      ! Envelope C6  114 x 162 mm
PAPER:ENV_C65             EQUATE(32)      ! Envelope C65 114 x 229 mm
PAPER:ENV_B4              EQUATE(33)      ! Envelope B4  250 x 353 mm
PAPER:ENV_B5              EQUATE(34)      ! Envelope B5  176 x 250 mm
PAPER:ENV_B6              EQUATE(35)      ! Envelope B6  176 x 125 mm
PAPER:ENV_ITALY           EQUATE(36)      ! Envelope 110 x 230 mm
PAPER:ENV_MONARCH         EQUATE(37)      ! Envelope Monarch 3.875 x 7.5 in
PAPER:ENV_PERSONAL        EQUATE(38)      ! 6 3/4 Envelope 3 5/8 x 6 1/2 in
```

```
PAPER:FANFOLD_US          EQUATE(39)      ! US Std Fanfold 14 7/8 x 11 in
PAPER:FANFOLD_STD_GERMAN  EQUATE(40)      ! German Std Fanfold 8 1/2 x 12 in
PAPER:FANFOLD_LGL_GERMAN  EQUATE(41)      ! German Legal Fanfold 8 1/2 x 13 in
PAPER:LAST                EQUATE(41)
PAPER:USER                EQUATE(256)

! File Driver Function equates for use with file{PROP:SupportsOp,DriverOp:n}

  ITEMIZE(1),PRE(DriverOp)
ADD                 EQUATE
BOF                 EQUATE
BUILDfile           EQUATE
APPEND              EQUATE
BUILDdyn            EQUATE
BUILDkey            EQUATE
CLOSE               EQUATE
COMMIT              EQUATE
COPY                EQUATE
CREATE              EQUATE
DELETE              EQUATE
DUPLICATE           EQUATE
EMPTY               EQUATE
EOF                 EQUATE
GETfilekey          EQUATE
GETfileptr          EQUATE
GETkeyptr           EQUATE
HOLD                EQUATE
LOCK                EQUATE(20)
LOGOUT              EQUATE(22)
NAME                EQUATE
NEXT                EQUATE
OPEN                EQUATE
PACK                EQUATE
POINTERfile         EQUATE
POINTERkey          EQUATE
FLUSH               EQUATE
PUT                 EQUATE
PREVIOUS            EQUATE
RECORDSfile         EQUATE
RECORDSkey          EQUATE
BUILDdynfilter      EQUATE
```

```
RELEASE          EQUATE(36)
REMOVE           EQUATE
RENAME           EQUATE
ROLLBACK         EQUATE(40)
SETfile          EQUATE
SETfilekey       EQUATE
SETfileptr       EQUATE
SETkey           EQUATE
SETkeykey        EQUATE
SETkeyptr        EQUATE
SETkeykeyptr     EQUATE
SHARE            EQUATE
SKIP             EQUATE
UNLOCK           EQUATE
ADDlen           EQUATE
BYTES            EQUATE
GETfileptrlen    EQUATE
PUTfileptr       EQUATE
PUTfileptrlen    EQUATE
STREAM           EQUATE
DUPLICATEkey     EQUATE
WATCH            EQUATE
APPENDlen        EQUATE
SEND             EQUATE
POSITIONfile     EQUATE
POSITIONkey      EQUATE
RESETfile        EQUATE
RESETkey         EQUATE
NOMEMO           EQUATE
REGETfile        EQUATE
REGETkey         EQUATE
NULL             EQUATE
SETNULL          EQUATE
SETNONNULL       EQUATE
SETproperty      EQUATE
GETproperty      EQUATE
GETblobdata      EQUATE(75)
PUTblobdata      EQUATE
BLOBSIZE         EQUATE
SETblobproperty  EQUATE
GETblobproperty  EQUATE
```

```
BUFFER           EQUATE
SETviewfields    EQUATE
CLEARfile        EQUATE
RESETviewfile    EQUATE
BUILDevent       EQUATE
SETkeyproperty   EQUATE
GETkeyproperty   EQUATE
DOproperty       EQUATE(88)
DOkeyproperty    EQUATE
DOblobproperty   EQUATE
VIEWSTART        EQUATE(92)
VIEWSTOP         EQUATE
GETNULLS         EQUATE(96)
S          EQUATE
GETSTATE         EQUATE
RESTORESTATE     EQUATE
CALLBACK         EQUATE
FREESTATE        EQUATE(102)
DESTROY          EQUATE(104)
  END

! Data Type Equates for use with file{PROP:SupportsType, DataType:n}

  ITEMIZE(1),PRE(DataType)
BYTE             EQUATE
SHORT            EQUATE
USHORT           EQUATE
DATE             EQUATE
TIME             EQUATE
LONG             EQUATE
ULONG            EQUATE
SREAL            EQUATE
REAL             EQUATE
DECIMAL          EQUATE
PDECIMAL         EQUATE
BFLOAT4          EQUATE(13)
BFLOAT8          EQUATE
STRING           EQUATE(18)
CSTRING          EQUATE
PSTRING          EQUATE
MEMO             EQUATE
```

```
BLOB              EQUATE(27)
  END




! These equates are to be used as the first parameter to the DELETEREG,
! GETREG and PUTREG statements

REG_CLASSES_ROOT      EQUATE(80000000h)
REG_CURRENT_USER      EQUATE(80000001h)
REG_LOCAL_MACHINE     EQUATE(80000002h)
REG_USERS             EQUATE(80000003h)
REG_PERFORMANCE_DATA  EQUATE(80000004h)
REG_CURRENT_CONFIG    EQUATE(80000005h)
REG_DYN_DATA          EQUATE(80000006h)
```

Template Equates

Template Defined Action Values

Template Defined Record Status Flags

File Access Modes

Constant field equate values for Toolbar browse control

Form VCR Request Codes

Tool bar navigation modes

Resize and Positional Strategies

Template Warnings

Browse Related Equates

Report Related Equates

```
! Template Equates

!Template Defined Action Values

InsertRecord EQUATE (1) ! Add a record to table
```

```
ChangeRecord EQUATE (2) ! Change the current record
DeleteRecord EQUATE (3) ! Delete the current record
SelectRecord EQUATE (4) ! Select the current record
ProcessRecord EQUATE (5) ! Process the current record
ViewRecord EQUATE (6) ! View the current record
RequestCompleted EQUATE (1) ! Update Completed
RequestCancelled EQUATE (2) ! Update Aborted

!Template Defined Record Status Flags

Record:OK EQUATE(0) ! Record passes range and filter
Record:OutOfRange EQUATE(1) ! Record fails range test
Record:Filtered EQUATE(2) ! Record fails filter tests

!File Access Modes

ReadOnly EQUATE (0H)
WriteOnly EQUATE (1H)
ReadWrite EQUATE (2H)

AnyAccess EQUATE (0H)
DenyAll EQUATE (10H)
DenyWrite EQUATE (20H)
DenyRead EQUATE (30H)
DenyNone EQUATE (40H)

!Constant field equate values for Toolbar browse control

TBarBrwFirst EQUATE(2000)
TBarBrwInsert EQUATE(TBarBrwFirst)
TBarBrwChange EQUATE(TBarBrwFirst+1)
TBarBrwDelete EQUATE(TBarBrwFirst+2)
TBarBrwSelect EQUATE(TBarBrwFirst+3)
TBarBrwBottom EQUATE(TBarBrwFirst+4)
TBarBrwTop EQUATE(TBarBrwFirst+5)
TBarBrwPageDown EQUATE(TBarBrwFirst+6)
TBarBrwPageUp EQUATE(TBarBrwFirst+7)
TBarBrwDown EQUATE(TBarBrwFirst+8)
TBarBrwUp EQUATE(TBarBrwFirst+9)
TBarBrwLocate EQUATE(TBarBrwFirst+10)
```

```
TBarBrwHistory EQUATE(TBarBrwFirst+11)
TBarBrwHelp EQUATE(TBarBrwFirst+12)
TBarBrwLast EQUATE(TBarBrwHelp)

!Form VCR Request Codes

VCRForward EQUATE(TBarBrwDown)
VCRBackward EQUATE(TBarBrwUp)
VCRPageForward EQUATE(TBarBrwPageDown)
VCRPageBackward EQUATE(TBarBrwPageUp)
VCRFirst EQUATE(TBarBrwTop)
VCRLast EQUATE(TBarBrwBottom)
VCRInsert EQUATE(TBarBrwInsert)
VCRNone EQUATE(0)

!Tool bar navigation modes

FormMode EQUATE(1)
BrowseMode EQUATE(2)
TreeMode EQUATE(3)

!Resize strategies

Resize:LockWidth EQUATE(01b) !locks width of control
Resize:LockHeight EQUATE(10b) !locks height of control

!Positional Strategies
!Horizontal
Resize:LockXPos EQUATE(0000000000000001b) !locks XPos of control

!fixes XPos offset of control to right of window:
Resize:FixRight EQUATE(0000000000000010b)

!fixes XPos offset of control to left of window:
Resize:FixLeft EQUATE(0000000000000100b)

!fixes XPos offset of control relative to centre
Resize:FixXCentre EQUATE(0000000000001000b)

!fixes to nearest window side - derived strategy:
```

```
Resize:FixNearestX EQUATE(0000000000010000b)



!Vertical
Resize:LockYPos EQUATE(0000000100000000b) !locks YPos of control

!fixes YPos offset of control to bottom of window
Resize:FixBottom EQUATE(0000001000000000b)

!fixes YPos offset of control to top of window:
Resize:FixTop EQUATE(0000010000000000b)

!fixes YPos offset of control relative to center:
Resize:FixYCentre EQUATE(0000100000000000b)

!fixes to nearest, window top of bottom = derived strategy
Resize:FixNearestY EQUATE(0001000000000000b)

!Application Resize Strategies

AppStrat:Resize EQUATE(1) !Resizes & moves all controls

!Resizes 'surface' controls moves others using Resize:FixNearestX+Resize:FixNearestY:
AppStrat:Spread EQUATE(2)

!Template Warnings

Warn:InvalidFile EQUATE (1)
Warn:InvalidKey EQUATE (2)
Warn:RebuildError EQUATE (3)
Warn:CreateError EQUATE (4)
Warn:CreateOpenError EQUATE (5)
Warn:ProcedureToDo EQUATE (6)
Warn:BadKeyedRec EQUATE (7)
Warn:OutOfRangeHigh EQUATE (8)
Warn:OutOfRangeLow EQUATE (9)
Warn:OutOfRange EQUATE (10)
Warn:NotInFile EQUATE (11)
Warn:RestrictUpdate EQUATE (12)
Warn:RestrictDelete EQUATE (13)
```

```
Warn:InsertError EQUATE (14)
Warn:RIUpdateError EQUATE (15)
Warn:UpdateError EQUATE (16)
Warn:RIDeleteError EQUATE (17)
Warn:DeleteError EQUATE (18)
Warn:InsertDisabled EQUATE (19)
Warn:UpdateDisabled EQUATE (20)
Warn:DeleteDisabled EQUATE (21)
Warn:NoCreate EQUATE (22)
Warn:ConfirmCancel EQUATE (23)
Warn:DuplicateKey EQUATE (24)
Warn:AutoIncError EQUATE (25)
Warn:FileLoadError EQUATE (26)
Warn:ConfirmCancelLoad EQUATE (27)
Warn:FileZeroLength EQUATE (28)
Warn:EndOfAsciiQueue EQUATE (29)
Warn:DiskError EQUATE (30)
Warn:ProcessActionError EQUATE (31)
Warn:StandardDelete EQUATE (32)
Warn:SaveOnCancel EQUATE (33)
Warn:LogoutError EQUATE (34)
Warn:RecordFetchError EQUATE (35)
Warn:ViewOpenError EQUATE (36)
Warn:NewRecordAdded EQUATE (37)
Warn:RIFormUpdateError EQUATE (38)



!Browse Related Equates
ScrollSort:Alpha EQUATE(' AFANATB BFBNBTC CFCNCT'|
&'D DFDNDTE EFENETF FFFNFT'|
&'G GFGNGTH HFHNHTI IFINIT'|
&'J JFJNJTK KFKNKTL LFLNLT'|
&'M MFMNMTN NFNNNTO OFONOT'|
&'P PFPNPTQ QNR RFRNRTS SF'|
&'SNSTT TFTNTTU UFUNUTV VF'|
&'VNVTW WFWNWTX XFXNXTY YF'|
&'YNYTZ ZN')

ScrollSort:Name EQUATE(' ALBAMEARNBAKBATBENBIABOBBRA'|
&'BROBUACACCARCENCHRCOECONCORCRU'|
&'DASDELDIADONDURELDEVEFELFISFLO'|
&'FREFUTGARGIBGOLGOSGREGUTHAMHEM'|
&'HOBHOTINGJASJONKAGKEAKIRKORKYO'|
&'LATLEOLIGLOUMACMAQMARMAUMCKMER'|
&'MILMONMORNATNOLOKEPAGPAUPETPIN'|
&'PORPULRAUREYROBROSRUBSALSCASCH'|
&'SCRSHASIGSKISNASOUSTESTISUNTAY'|
&'TIRTUCVANWACWASWEIWIEWIMWOLYOR')

ScrollSort:AllowAlpha EQUATE(1)
ScrollSort:AllowAlt EQUATE(2)
ScrollSort:AllowNumeric EQUATE(4)
ScrollSort:CaseSensitive EQUATE(8)

SortRequest:SelectSort EQUATE(1)
SortRequest:Reset EQUATE(2)
SortRequest:LocateRecord EQUATE(3)

SortResult:Changed EQUATE(1)
SortResult:OK EQUATE(2)
LocateOnPosition EQUATE(1)
LocateOnValue EQUATE(2)
LocateOnEdit EQUATE(3)
FillBackward EQUATE(1)
FillForward EQUATE(2)
RefreshOnPosition EQUATE(1)
RefreshOnQueue EQUATE(2)
RefreshOnTop EQUATE(3)
RefreshOnBottom EQUATE(4)
RefreshOnCurrent EQUATE(5)



!Report Related Equates
EVENT:Preview:Print EQUATE (401H)
EVENT:Preview:Cancel EQUATE (402H)
EVENT:Preview:Zoom EQUATE (403H)
EVENT:Preview:NextPage EQUATE (404H)
EVENT:Preview:PrevPage EQUATE (405H)
EVENT:Preview:Jump EQUATE (406h)
EVENT:Preview:ChangeDisplay EQUATE (407H)
EVENT:Preview:DisableNext EQUATE (450h)
EVENT:Preview:EnableNext EQUATE (451h)
```

```
EVENT:Preview:DisablePrev EQUATE (452h)

EVENT:Preview:EnablePrev EQUATE (453h)

EVENT:Preview:DirectZoom EQUATE (454h)

EVENT:Preview:DirectUnzoom EQUATE (455h)


Preview:OutOfPagesText EQUATE ('There are no more pages to display')

Preview:OutOfPagesHead EQUATE ('End of Report')

Preview:DisplayText EQUATE (1)

Preview:DisplayIcons EQUATE (2)

Preview:DisplayAll EQUATE (3)
```

Keycode Equates List

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

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

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

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

```
Key0 EQUATE(0030H) !0 Key
Key1 EQUATE(0031H) !1 Key
Key2 EQUATE(0032H) !2 Key
Key3 EQUATE(0033H) !3 Key
Key4 EQUATE(0034H) !4 Key
Key5 EQUATE(0035H) !5 Key
Key6 EQUATE(0036H) !6 Key
Key7 EQUATE(0037H) !7 Key
Key8 EQUATE(0038H) !8 Key
Key9 EQUATE(0039H) !9 Key

AKey EQUATE(0041H) !A Key
BKey EQUATE(0042H) !B Key
CKey EQUATE(0043H) !C Key
DKey EQUATE(0044H) !D Key
EKey EQUATE(0045H) !E Key
FKey EQUATE(0046H) !F Key
GKey EQUATE(0047H) !G Key
HKey EQUATE(0048H) !H Key
IKey EQUATE(0049H) !I Key
JKey EQUATE(004AH) !J Key
KKey EQUATE(004BH) !K Key
LKey EQUATE(004CH) !L Key
MKey EQUATE(004DH) !M Key
NKey EQUATE(004EH) !N Key
OKey EQUATE(004FH) !O Key
PKey EQUATE(0050H) !P Key
QKey EQUATE(0051H) !Q Key
RKey EQUATE(0052H) !R Key
```

```
SKey EQUATE(0053H) !S Key
TKey EQUATE(0054H) !T Key
UKey EQUATE(0055H) !U Key
VKey EQUATE(0056H) !V Key
WKey EQUATE(0057H) !W Key
XKey EQUATE(0058H) !X Key
YKey EQUATE(0059H) !Y Key
ZKey EQUATE(005AH) !Z Key

F1Key EQUATE(0070H) !F1 Key
F2Key EQUATE(0071H) !F2 Key
F3Key EQUATE(0072H) !F3 Key
F4Key EQUATE(0073H) !F4 Key
F5Key EQUATE(0074H) !F5 Key
F6Key EQUATE(0075H) !F6 Key
F7Key EQUATE(0076H) !F7 Key
F8Key EQUATE(0077H) !F8 Key
F9Key EQUATE(0078H) !F9 Key
F10Key EQUATE(0079H) !F10 Key
F11Key EQUATE(007AH) !F11 Key
F12Key EQUATE(007BH) !F12 Key

AstKey EQUATE(006AH) !Asterisk Key
BSKey EQUATE(0008H) !Backspace Key
CapsLockKey EQUATE(0014H) !CapsLock Key
DecimalKey EQUATE(006EH) !Decimal Key
DeleteKey EQUATE(002EH) !Delete Key
DownKey EQUATE(0028H) !Cursor Down Key
EndKey EQUATE(0023H) !End Key
EnterKey EQUATE(000DH) !Enter Key
EscKey EQUATE(001BH) !Esc Key
HomeKey EQUATE(0024H) !Home Key
InsertKey EQUATE(002DH) !Insert Key
LeftKey EQUATE(0025H) !Cursor Left Key
MinusKey EQUATE(006DH) !Minus Key
PauseKey EQUATE(0013H) !Pause Key
PgDnKey EQUATE(0022H) !PgDn Key
PgUpKey EQUATE(0021H) !PgUp Key
PlusKey EQUATE(006BH) !Plus Key
PrintKey EQUATE(002CH) !PrintScreen Key
RightKey EQUATE(0027H) !Cursor Right Key
```

```
SlashKey EQUATE(006FH) !Slash Key
SpaceKey EQUATE(0020H) !Spacebar
TabKey EQUATE(0009H) !Tab Key
UpKey EQUATE(0026H) !Cursor Up Key

KeyPad0 EQUATE(0060H) !0 on numeric keypad
KeyPad1 EQUATE(0061H) !1 on numeric keypad
KeyPad2 EQUATE(0062H) !2 on numeric keypad
KeyPad3 EQUATE(0063H) !3 on numeric keypad
KeyPad4 EQUATE(0064H) !4 on numeric keypad
KeyPad5 EQUATE(0065H) !5 on numeric keypad
KeyPad6 EQUATE(0066H) !6 on numeric keypad
KeyPad7 EQUATE(0067H) !7 on numeric keypad
KeyPad8 EQUATE(0068H) !8 on numeric keypad
KeyPad9 EQUATE(0069H) !9 on numeric keypad

MouseLeft EQUATE(0001H) !Left mouse button
MouseRight EQUATE(0002H) !Right mouse button
MouseCenter EQUATE(0003H) !Middle mouse button
MouseLeft2 EQUATE(0005H) !Left mouse double-click
MouseRight2 EQUATE(0006H) !Right mouse double-click
MouseCenter2 EQUATE(0007H) !Middle mouse double-click
MouseLeftUp EQUATE(0081H) !Left mouse button release
MouseRightUp EQUATE(0082H) !Right mouse button release
MouseCenterUp EQUATE(0083H) !Middle mouse button release

Alt0 EQUATE(0430H) !Alt-0 Key
Alt1 EQUATE(0431H) !Alt-1 Key
Alt2 EQUATE(0432H) !Alt-2 Key
Alt3 EQUATE(0433H) !Alt-3 Key
Alt4 EQUATE(0434H) !Alt-4 Key
Alt5 EQUATE(0435H) !Alt-5 Key
Alt6 EQUATE(0436H) !Alt-6 Key
Alt7 EQUATE(0437H) !Alt-7 Key
Alt8 EQUATE(0438H) !Alt-8 Key
Alt9 EQUATE(0439H) !Alt-9 Key

AltA EQUATE(0441H) !Alt-A Key
AltB EQUATE(0442H) !Alt-B Key
AltC EQUATE(0443H) !Alt-C Key
AltD EQUATE(0444H) !Alt-D Key
```

```
AltE EQUATE(0445H) !Alt-E Key
AltF EQUATE(0446H) !Alt-F Key
AltG EQUATE(0447H) !Alt-G Key
AltH EQUATE(0448H) !Alt-H Key
AltI EQUATE(0449H) !Alt-I Key
AltJ EQUATE(044AH) !Alt-J Key
AltK EQUATE(044BH) !Alt-K Key
AltL EQUATE(044CH) !Alt-L Key
AltM EQUATE(044DH) !Alt-M Key
AltN EQUATE(044EH) !Alt-N Key
AltO EQUATE(044FH) !Alt-O Key
AltP EQUATE(0450H) !Alt-P Key
AltQ EQUATE(0451H) !Alt-Q Key
AltR EQUATE(0452H) !Alt-R Key
AltS EQUATE(0453H) !Alt-S Key
AltT EQUATE(0454H) !Alt-T Key
AltU EQUATE(0455H) !Alt-U Key
AltV EQUATE(0456H) !Alt-V Key
AltW EQUATE(0457H) !Alt-W Key
AltX EQUATE(0458H) !Alt-X Key
AltY EQUATE(0459H) !Alt-Y Key
AltZ EQUATE(045AH) !Alt-Z Key

AltF1 EQUATE(0470H) !Alt-F1 Key
AltF2 EQUATE(0471H) !Alt-F2 Key
AltF3 EQUATE(0472H) !Alt-F3 Key
AltF4 EQUATE(0473H) !Alt-F4 Key
AltF5 EQUATE(0474H) !Alt-F5 Key
AltF6 EQUATE(0475H) !Alt-F6 Key
AltF7 EQUATE(0476H) !Alt-F7 Key
AltF8 EQUATE(0477H) !Alt-F8 Key
AltF9 EQUATE(0478H) !Alt-F9 Key
AltF10 EQUATE(0479H) !Alt-F10 Key
AltF11 EQUATE(047AH) !Alt-F11 Key
AltF12 EQUATE(047BH) !Alt-F12 Key

AltAst EQUATE(046AH) !Alt-Asterisk Key
AltBS EQUATE(0408H) !Alt-Backspace Key
AltDecimal EQUATE(046EH) !Alt-Decimal Key
AltDelete EQUATE(042EH) !Alt-Delete Key
AltDown EQUATE(0428H) !Alt-Cursor Down Key
```

```
AltEnd EQUATE(0423H) !Alt-End Key
AltEnter EQUATE(040DH) !Alt-Enter Key
AltEsc EQUATE(041BH) !Alt-Esc Key
AltHome EQUATE(0424H) !Alt-Home Key
AltInsert EQUATE(042DH) !Alt-Insert Key
AltLeft EQUATE(0425H) !Alt-Cursor Left Key
AltMinus EQUATE(046DH) !Alt-Minus Key
AltPause EQUATE(0413H) !Alt-Pause Key
AltPgDn EQUATE(0422H) !Alt-PgDn Key
AltPgUp EQUATE(0421H) !Alt-PgUp Key
AltPlus EQUATE(046BH) !Alt-Plus Key
AltPrint EQUATE(042CH) !Alt-PrintScreen Key
AltRight EQUATE(0427H) !Alt-Cursor Right Key
AltSlash EQUATE(046FH) !Alt-Slash Key
AltSpace EQUATE(0420H) !Alt-Spacebar
AltTab EQUATE(0409H) !Alt-Tab Key
AltUp EQUATE(0426H) !Alt-Cursor Up Key

AltPad0 EQUATE(0460H) !Alt-0 on numeric keypad
AltPad1 EQUATE(0461H) !Alt-1 on numeric keypad
AltPad2 EQUATE(0462H) !Alt-2 on numeric keypad
AltPad3 EQUATE(0463H) !Alt-3 on numeric keypad
AltPad4 EQUATE(0464H) !Alt-4 on numeric keypad
AltPad5 EQUATE(0465H) !Alt-5 on numeric keypad
AltPad6 EQUATE(0466H) !Alt-6 on numeric keypad
AltPad7 EQUATE(0467H) !Alt-7 on numeric keypad
AltPad8 EQUATE(0468H) !Alt-8 on numeric keypad
AltPad9 EQUATE(0469H) !Alt-9 on numeric keypad

AltMouseLeft EQUATE(0401H) !Alt-Left mouse button
AltMouseRight EQUATE(0402H) !Alt-Right mouse button
AltMouseCenter EQUATE(0403H) !Alt-Middle mouse button
AltMouseLeft2 EQUATE(0405H) !Alt-Left mouse double-click
AltMouseRight2 EQUATE(0406H) !Alt-Right mouse double-click
AltMouseCenter2 EQUATE(0407H) !Alt-Middle mouse double-click

Ctrl0 EQUATE(0230H) !Ctrl-0 Key
Ctrl1 EQUATE(0231H) !Ctrl-1 Key
Ctrl2 EQUATE(0232H) !Ctrl-2 Key
Ctrl3 EQUATE(0233H) !Ctrl-3 Key
Ctrl4 EQUATE(0234H) !Ctrl-4 Key
```

```
Ctrl5 EQUATE(0235H) !Ctrl-5 Key
Ctrl6 EQUATE(0236H) !Ctrl-6 Key
Ctrl7 EQUATE(0237H) !Ctrl-7 Key
Ctrl8 EQUATE(0238H) !Ctrl-8 Key
Ctrl9 EQUATE(0239H) !Ctrl-9 Key

CtrlA EQUATE(0241H) !Ctrl-A Key
CtrlB EQUATE(0242H) !Ctrl-B Key
CtrlC EQUATE(0243H) !Ctrl-C Key
CtrlD EQUATE(0244H) !Ctrl-D Key
CtrlE EQUATE(0245H) !Ctrl-E Key
CtrlF EQUATE(0246H) !Ctrl-F Key
CtrlG EQUATE(0247H) !Ctrl-G Key
CtrlH EQUATE(0248H) !Ctrl-H Key
CtrlI EQUATE(0249H) !Ctrl-I Key
CtrlJ EQUATE(024AH) !Ctrl-J Key
CtrlK EQUATE(024BH) !Ctrl-K Key
CtrlL EQUATE(024CH) !Ctrl-L Key
CtrlM EQUATE(024DH) !Ctrl-M Key
CtrlN EQUATE(024EH) !Ctrl-N Key
CtrlO EQUATE(024FH) !Ctrl-O Key
CtrlP EQUATE(0250H) !Ctrl-P Key
CtrlQ EQUATE(0251H) !Ctrl-Q Key
CtrlR EQUATE(0252H) !Ctrl-R Key
CtrlS EQUATE(0253H) !Ctrl-S Key
CtrlT EQUATE(0254H) !Ctrl-T Key
CtrlU EQUATE(0255H) !Ctrl-U Key
CtrlV EQUATE(0256H) !Ctrl-V Key
CtrlW EQUATE(0257H) !Ctrl-W Key
CtrlX EQUATE(0258H) !Ctrl-X Key
CtrlY EQUATE(0259H) !Ctrl-Y Key
CtrlZ EQUATE(025AH) !Ctrl-Z Key

CtrlF1 EQUATE(0270H) !Ctrl-F1 Key
CtrlF2 EQUATE(0271H) !Ctrl-F2 Key
CtrlF3 EQUATE(0272H) !Ctrl-F3 Key
CtrlF4 EQUATE(0273H) !Ctrl-F4 Key
CtrlF5 EQUATE(0274H) !Ctrl-F5 Key
CtrlF6 EQUATE(0275H) !Ctrl-F6 Key
CtrlF7 EQUATE(0276H) !Ctrl-F7 Key
CtrlF8 EQUATE(0277H) !Ctrl-F8 Key
```

```
CtrlF9 EQUATE(0278H) !Ctrl-F9 Key
CtrlF10 EQUATE(0279H) !Ctrl-F10 Key
CtrlF11 EQUATE(027AH) !Ctrl-F11 Key
CtrlF12 EQUATE(027BH) !Ctrl-F12 Key

CtrlAst EQUATE(026AH) !Ctrl-Asterisk Key
CtrlBS EQUATE(0208H) !Ctrl-Backspace Key
CtrlDecimal EQUATE(026EH) !Ctrl-Decimal Key
CtrlDelete EQUATE(022EH) !Ctrl-Delete Key
CtrlDown EQUATE(0228H) !Ctrl-Cursor Down Key
CtrlEnd EQUATE(0223H) !Ctrl-End Key
CtrlEnter EQUATE(020DH) !Ctrl-Enter Key
CtrlEsc EQUATE(021BH) !Ctrl-Esc Key
CtrlHome EQUATE(0224H) !Ctrl-Home Key
CtrlInsert EQUATE(022DH) !Ctrl-Insert Key
CtrlLeft EQUATE(0225H) !Ctrl-Cursor Left Key
CtrlMinus EQUATE(026DH) !Ctrl-Minus Key
CtrlPause EQUATE(0213H) !Ctrl-Pause Key
CtrlPgDn EQUATE(0222H) !Ctrl-PgDn Key
CtrlPgUp EQUATE(0221H) !Ctrl-PgUp Key
CtrlPlus EQUATE(026BH) !Ctrl-Plus Key
CtrlPrint EQUATE(022CH) !Ctrl-PrintScreen Key
CtrlRight EQUATE(0227H) !Ctrl-Cursor Right Key
CtrlSlash EQUATE(026FH) !Ctrl-Slash Key
CtrlSpace EQUATE(0220H) !Ctrl-Spacebar
CtrlTab EQUATE(0209H) !Ctrl-Tab Key
CtrlUp EQUATE(0226H) !Ctrl-Cursor Up Key

CtrlPad0 EQUATE(0260H) !Ctrl-0 on numeric keypad
CtrlPad1 EQUATE(0261H) !Ctrl-1 on numeric keypad
CtrlPad2 EQUATE(0262H) !Ctrl-2 on numeric keypad
CtrlPad3 EQUATE(0263H) !Ctrl-3 on numeric keypad
CtrlPad4 EQUATE(0264H) !Ctrl-4 on numeric keypad
CtrlPad5 EQUATE(0265H) !Ctrl-5 on numeric keypad
CtrlPad6 EQUATE(0266H) !Ctrl-6 on numeric keypad
CtrlPad7 EQUATE(0267H) !Ctrl-7 on numeric keypad
CtrlPad8 EQUATE(0268H) !Ctrl-8 on numeric keypad
CtrlPad9 EQUATE(0269H) !Ctrl-9 on numeric keypad

CtrlMouseLeft EQUATE(0201H) !Ctrl-Left mouse button
CtrlMouseRight EQUATE(0202H) !Ctrl-Right mouse button
```

```
CtrlMouseCenter EQUATE(0203H) !Ctrl-Middle mouse button
CtrlMouseLeft2 EQUATE(0205H) !Ctrl-Left mouse double-click
CtrlMouseRight2 EQUATE(0206H) !Ctrl-Right mouse double-click
CtrlMouseCenter2 EQUATE(0207H) !Ctrl-Middle mouse double-click

Shift0 EQUATE(0130H) !Shift-0 Key
Shift1 EQUATE(0131H) !Shift-1 Key
Shift2 EQUATE(0132H) !Shift-2 Key
Shift3 EQUATE(0133H) !Shift-3 Key
Shift4 EQUATE(0134H) !Shift-4 Key
Shift5 EQUATE(0135H) !Shift-5 Key
Shift6 EQUATE(0136H) !Shift-6 Key
Shift7 EQUATE(0137H) !Shift-7 Key
Shift8 EQUATE(0138H) !Shift-8 Key
Shift9 EQUATE(0139H) !Shift-9 Key

ShiftA EQUATE(0141H) !Shift-A Key
ShiftB EQUATE(0142H) !Shift-B Key
ShiftC EQUATE(0143H) !Shift-C Key
ShiftD EQUATE(0144H) !Shift-D Key
ShiftE EQUATE(0145H) !Shift-E Key
ShiftF EQUATE(0146H) !Shift-F Key
ShiftG EQUATE(0147H) !Shift-G Key
ShiftH EQUATE(0148H) !Shift-H Key
ShiftI EQUATE(0149H) !Shift-I Key
ShiftJ EQUATE(014AH) !Shift-J Key
ShiftK EQUATE(014BH) !Shift-K Key
ShiftL EQUATE(014CH) !Shift-L Key
ShiftM EQUATE(014DH) !Shift-M Key
ShiftN EQUATE(014EH) !Shift-N Key
ShiftO EQUATE(014FH) !Shift-O Key
ShiftP EQUATE(0150H) !Shift-P Key
ShiftQ EQUATE(0151H) !Shift-Q Key
ShiftR EQUATE(0152H) !Shift-R Key
ShiftS EQUATE(0153H) !Shift-S Key
ShiftT EQUATE(0154H) !Shift-T Key
ShiftU EQUATE(0155H) !Shift-U Key
ShiftV EQUATE(0156H) !Shift-V Key
ShiftW EQUATE(0157H) !Shift-W Key
ShiftX EQUATE(0158H) !Shift-X Key
ShiftY EQUATE(0159H) !Shift-Y Key
```

```
ShiftZ EQUATE(015AH) !Shift-Z Key


ShiftF1 EQUATE(0170H) !Shift-F1 Key
ShiftF2 EQUATE(0171H) !Shift-F2 Key
ShiftF3 EQUATE(0172H) !Shift-F3 Key
ShiftF4 EQUATE(0173H) !Shift-F4 Key
ShiftF5 EQUATE(0174H) !Shift-F5 Key
ShiftF6 EQUATE(0175H) !Shift-F6 Key
ShiftF7 EQUATE(0176H) !Shift-F7 Key
ShiftF8 EQUATE(0177H) !Shift-F8 Key
ShiftF9 EQUATE(0178H) !Shift-F9 Key
ShiftF10 EQUATE(0179H) !Shift-F10 Key
ShiftF11 EQUATE(017AH) !Shift-F11 Key
ShiftF12 EQUATE(017BH) !Shift-F12 Key


ShiftAst EQUATE(016AH) !Shift-Asterisk Key
ShiftBS EQUATE(0108H) !Shift-Backspace Key
ShiftDecimal EQUATE(016EH) !Shift-Decimal Key
ShiftDelete EQUATE(012EH) !Shift-Delete Key
ShiftDown EQUATE(0128H) !Shift-Cursor Down Key
ShiftEnd EQUATE(0123H) !Shift-End Key
ShiftEnter EQUATE(010DH) !Shift-Enter Key
ShiftEsc EQUATE(011BH) !Shift-Esc Key
ShiftHome EQUATE(0124H) !Shift-Home Key
ShiftInsert EQUATE(012DH) !Shift-Insert Key
ShiftLeft EQUATE(0125H) !Shift-Cursor Left Key
ShiftMinus EQUATE(016DH) !Shift-Minus Key
ShiftPause EQUATE(0113H) !Shift-Pause Key
ShiftPgDn EQUATE(0122H) !Shift-PgDn Key
ShiftPgUp EQUATE(0121H) !Shift-PgUp Key
ShiftPlus EQUATE(016BH) !Shift-Plus Key
ShiftPrint EQUATE(012CH) !Shift-PrintScreen Key
ShiftRight EQUATE(0127H) !Shift-Cursor Right Key
ShiftSlash EQUATE(016FH) !Shift-Slash Key
ShiftSpace EQUATE(0120H) !Shift-Spacebar
ShiftTab EQUATE(0109H) !Shift-Tab Key
ShiftUp EQUATE(0126H) !Shift-Cursor Up Key


ShiftPad0 EQUATE(0160H) !Shift-0 on numeric keypad
ShiftPad1 EQUATE(0161H) !Shift-1 on numeric keypad
ShiftPad2 EQUATE(0162H) !Shift-2 on numeric keypad
```

```
ShiftPad3 EQUATE(0163H) !Shift-3 on numeric keypad
ShiftPad4 EQUATE(0164H) !Shift-4 on numeric keypad
ShiftPad5 EQUATE(0165H) !Shift-5 on numeric keypad
ShiftPad6 EQUATE(0166H) !Shift-6 on numeric keypad
ShiftPad7 EQUATE(0167H) !Shift-7 on numeric keypad
ShiftPad8 EQUATE(0168H) !Shift-8 on numeric keypad
ShiftPad9 EQUATE(0169H) !Shift-9 on numeric keypad



ShiftMouseLeft EQUATE(0101H) !Shift-Left mouse button
ShiftMouseRight EQUATE(0102H) !Shift-Right mouse button
ShiftMouseCenter EQUATE(0103H) !Shift-Middle mouse button
ShiftMouseLeft2 EQUATE(0105H) !Shift-Left mouse double-click
ShiftMouseRight2 EQUATE(0106H) !Shift-Right mouse double-click
ShiftMouseCenter2 EQUATE(0107H) !Shift-Middle mouse double-click

AltShift0 EQUATE(0530H) !Alt-Shift-0 Key
AltShift1 EQUATE(0531H) !Alt-Shift-1 Key
AltShift2 EQUATE(0532H) !Alt-Shift-2 Key
AltShift3 EQUATE(0533H) !Alt-Shift-3 Key
AltShift4 EQUATE(0534H) !Alt-Shift-4 Key
AltShift5 EQUATE(0535H) !Alt-Shift-5 Key
AltShift6 EQUATE(0536H) !Alt-Shift-6 Key
AltShift7 EQUATE(0537H) !Alt-Shift-7 Key
AltShift8 EQUATE(0538H) !Alt-Shift-8 Key
AltShift9 EQUATE(0539H) !Alt-Shift-9 Key

AltShiftA EQUATE(0541H) !Alt-Shift-A Key
AltShiftB EQUATE(0542H) !Alt-Shift-B Key
AltShiftC EQUATE(0543H) !Alt-Shift-C Key
AltShiftD EQUATE(0544H) !Alt-Shift-D Key
AltShiftE EQUATE(0545H) !Alt-Shift-E Key
AltShiftF EQUATE(0546H) !Alt-Shift-F Key
AltShiftG EQUATE(0547H) !Alt-Shift-G Key
AltShiftH EQUATE(0548H) !Alt-Shift-H Key
AltShiftI EQUATE(0549H) !Alt-Shift-I Key
AltShiftJ EQUATE(054AH) !Alt-Shift-J Key
AltShiftK EQUATE(054BH) !Alt-Shift-K Key
AltShiftL EQUATE(054CH) !Alt-Shift-L Key
AltShiftM EQUATE(054DH) !Alt-Shift-M Key
AltShiftN EQUATE(054EH) !Alt-Shift-N Key
```

```
AltShiftO EQUATE(054FH) !Alt-Shift-O Key
AltShiftP EQUATE(0550H) !Alt-Shift-P Key
AltShiftQ EQUATE(0551H) !Alt-Shift-Q Key
AltShiftR EQUATE(0552H) !Alt-Shift-R Key
AltShiftS EQUATE(0553H) !Alt-Shift-S Key
AltShiftT EQUATE(0554H) !Alt-Shift-T Key
AltShiftU EQUATE(0555H) !Alt-Shift-U Key
AltShiftV EQUATE(0556H) !Alt-Shift-V Key
AltShiftW EQUATE(0557H) !Alt-Shift-W Key
AltShiftX EQUATE(0558H) !Alt-Shift-X Key
AltShiftY EQUATE(0559H) !Alt-Shift-Y Key
AltShiftZ EQUATE(055AH) !Alt-Shift-Z Key

AltShiftF1 EQUATE(0570H) !Alt-Shift-F1 Key
AltShiftF2 EQUATE(0571H) !Alt-Shift-F2 Key
AltShiftF3 EQUATE(0572H) !Alt-Shift-F3 Key
AltShiftF4 EQUATE(0573H) !Alt-Shift-F4 Key
AltShiftF5 EQUATE(0574H) !Alt-Shift-F5 Key
AltShiftF6 EQUATE(0575H) !Alt-Shift-F6 Key
AltShiftF7 EQUATE(0576H) !Alt-Shift-F7 Key
AltShiftF8 EQUATE(0577H) !Alt-Shift-F8 Key
AltShiftF9 EQUATE(0578H) !Alt-Shift-F9 Key
AltShiftF10 EQUATE(0579H) !Alt-Shift-F10 Key
AltShiftF11 EQUATE(057AH) !Alt-Shift-F11 Key
AltShiftF12 EQUATE(057BH) !Alt-Shift-F12 Key

AltShiftAst EQUATE(056AH) !Alt-Shift-Asterisk Key
AltShiftBS EQUATE(0508H) !Alt-Shift-Backspace Key
AltShiftDecimal EQUATE(056EH) !Alt-Shift-Decimal Key
AltShiftDelete EQUATE(052EH) !Alt-Shift-Delete Key
AltShiftDown EQUATE(0528H) !Alt-Shift-Cursor Down Key
AltShiftEnd EQUATE(0523H) !Alt-Shift-End Key
AltShiftEnter EQUATE(050DH) !Alt-Shift-Enter Key
AltShiftEsc EQUATE(051BH) !Alt-Shift-Esc Key
AltShiftHome EQUATE(0524H) !Alt-Shift-Home Key
AltShiftInsert EQUATE(052DH) !Alt-Shift-Insert Key
AltShiftLeft EQUATE(0525H) !Alt-Shift-Cursor Left Key
AltShiftMinus EQUATE(056DH) !Alt-Shift-Minus Key
AltShiftPause EQUATE(0513H) !Alt-Shift-Pause Key
AltShiftPgDn EQUATE(0522H) !Alt-Shift-PgDn Key
AltShiftPgUp EQUATE(0521H) !Alt-Shift-PgUp Key
```

```
AltShiftPlus EQUATE(056BH) !Alt-Shift-Plus Key
AltShiftPrint EQUATE(052CH) !Alt-Shift-PrintScreen Key
AltShiftRight EQUATE(0527H) !Alt-Shift-Cursor Right Key
AltShiftSlash EQUATE(056FH) !Alt-Shift-Slash Key
AltShiftSpace EQUATE(0520H) !Alt-Shift-Spacebar
AltShiftTab EQUATE(0509H) !Alt-Shift-Tab Key
AltShiftUp EQUATE(0526H) !Alt-Shift-Cursor Up Key

AltShiftPad0 EQUATE(0560H) !Alt-Shift-0 on numeric keypad
AltShiftPad1 EQUATE(0561H) !Alt-Shift-1 on numeric keypad
AltShiftPad2 EQUATE(0562H) !Alt-Shift-2 on numeric keypad
AltShiftPad3 EQUATE(0563H) !Alt-Shift-3 on numeric keypad
AltShiftPad4 EQUATE(0564H) !Alt-Shift-4 on numeric keypad
AltShiftPad5 EQUATE(0565H) !Alt-Shift-5 on numeric keypad
AltShiftPad6 EQUATE(0566H) !Alt-Shift-6 on numeric keypad
AltShiftPad7 EQUATE(0567H) !Alt-Shift-7 on numeric keypad
AltShiftPad8 EQUATE(0568H) !Alt-Shift-8 on numeric keypad
AltShiftPad9 EQUATE(0569H) !Alt-Shift-9 on numeric keypad


AltShiftMouseLeft EQUATE(0501H) !Alt-Shift-Left mouse button
AltShiftMouseRight EQUATE(0502H) !Alt-Shift-Right mouse button
AltShiftMouseCenter EQUATE(0503H) !Alt-Shift-Middle mouse button
AltShiftMouseLeft2 EQUATE(0505H) !Alt-Shift-Left mouse double-click
AltShiftMouseRight2 EQUATE(0506H) !Alt-Shift-Right mouse double-click
AltShiftMouseCenter2 EQUATE(0507H) !Alt-Shift-Middle mouse double-click

CtrlShift0 EQUATE(0330H) !Ctrl-Shift-0 Key
CtrlShift1 EQUATE(0331H) !Ctrl-Shift-1 Key
CtrlShift2 EQUATE(0332H) !Ctrl-Shift-2 Key
CtrlShift3 EQUATE(0333H) !Ctrl-Shift-3 Key
CtrlShift4 EQUATE(0334H) !Ctrl-Shift-4 Key
CtrlShift5 EQUATE(0335H) !Ctrl-Shift-5 Key
CtrlShift6 EQUATE(0336H) !Ctrl-Shift-6 Key
CtrlShift7 EQUATE(0337H) !Ctrl-Shift-7 Key
CtrlShift8 EQUATE(0338H) !Ctrl-Shift-8 Key
CtrlShift9 EQUATE(0339H) !Ctrl-Shift-9 Key

CtrlShiftA EQUATE(0341H) !Ctrl-Shift-A Key
CtrlShiftB EQUATE(0342H) !Ctrl-Shift-B Key
CtrlShiftC EQUATE(0343H) !Ctrl-Shift-C Key
```

```
CtrlShiftD EQUATE(0344H) !Ctrl-Shift-D Key
CtrlShiftE EQUATE(0345H) !Ctrl-Shift-E Key
CtrlShiftF EQUATE(0346H) !Ctrl-Shift-F Key
CtrlShiftG EQUATE(0347H) !Ctrl-Shift-G Key
CtrlShiftH EQUATE(0348H) !Ctrl-Shift-H Key
CtrlShiftI EQUATE(0349H) !Ctrl-Shift-I Key
CtrlShiftJ EQUATE(034AH) !Ctrl-Shift-J Key
CtrlShiftK EQUATE(034BH) !Ctrl-Shift-K Key
CtrlShiftL EQUATE(034CH) !Ctrl-Shift-L Key
CtrlShiftM EQUATE(034DH) !Ctrl-Shift-M Key
CtrlShiftN EQUATE(034EH) !Ctrl-Shift-N Key
CtrlShiftO EQUATE(034FH) !Ctrl-Shift-O Key
CtrlShiftP EQUATE(0350H) !Ctrl-Shift-P Key
CtrlShiftQ EQUATE(0351H) !Ctrl-Shift-Q Key
CtrlShiftR EQUATE(0352H) !Ctrl-Shift-R Key
CtrlShiftS EQUATE(0353H) !Ctrl-Shift-S Key
CtrlShiftT EQUATE(0354H) !Ctrl-Shift-T Key
CtrlShiftU EQUATE(0355H) !Ctrl-Shift-U Key
CtrlShiftV EQUATE(0356H) !Ctrl-Shift-V Key
CtrlShiftW EQUATE(0357H) !Ctrl-Shift-W Key
CtrlShiftX EQUATE(0358H) !Ctrl-Shift-X Key
CtrlShiftY EQUATE(0359H) !Ctrl-Shift-Y Key
CtrlShiftZ EQUATE(035AH) !Ctrl-Shift-Z Key

CtrlShiftF1 EQUATE(0370H) !Ctrl-Shift-F1 Key
CtrlShiftF2 EQUATE(0371H) !Ctrl-Shift-F2 Key
CtrlShiftF3 EQUATE(0372H) !Ctrl-Shift-F3 Key
CtrlShiftF4 EQUATE(0373H) !Ctrl-Shift-F4 Key
CtrlShiftF5 EQUATE(0374H) !Ctrl-Shift-F5 Key
CtrlShiftF6 EQUATE(0375H) !Ctrl-Shift-F6 Key
CtrlShiftF7 EQUATE(0376H) !Ctrl-Shift-F7 Key
CtrlShiftF8 EQUATE(0377H) !Ctrl-Shift-F8 Key
CtrlShiftF9 EQUATE(0378H) !Ctrl-Shift-F9 Key
CtrlShiftF10 EQUATE(0379H) !Ctrl-Shift-F10 Key
CtrlShiftF11 EQUATE(037AH) !Ctrl-Shift-F11 Key
CtrlShiftF12 EQUATE(037BH) !Ctrl-Shift-F12 Key

CtrlShiftAst EQUATE(036AH) !Ctrl-Shift-Asterisk Key
CtrlShiftBS EQUATE(0308H) !Ctrl-Shift-Backspace Key
CtrlShiftDecimal EQUATE(036EH) !Ctrl-Shift-Decimal Key
CtrlShiftDelete EQUATE(032EH) !Ctrl-Shift-Delete Key
```

```
CtrlShiftDown EQUATE(0328H) !Ctrl-Shift-Cursor Down Key
CtrlShiftEnd EQUATE(0323H) !Ctrl-Shift-End Key
CtrlShiftEnter EQUATE(030DH) !Ctrl-Shift-Enter Key
CtrlShiftEsc EQUATE(031BH) !Ctrl-Shift-Esc Key
CtrlShiftHome EQUATE(0324H) !Ctrl-Shift-Home Key
CtrlShiftInsert EQUATE(032DH) !Ctrl-Shift-Insert Key
CtrlShiftLeft EQUATE(0325H) !Ctrl-Shift-Cursor Left Key
CtrlShiftMinus EQUATE(036DH) !Ctrl-Shift-Minus Key
CtrlShiftPause EQUATE(0313H) !Ctrl-Shift-Pause Key
CtrlShiftPgDn EQUATE(0322H) !Ctrl-Shift-PgDn Key
CtrlShiftPgUp EQUATE(0321H) !Ctrl-Shift-PgUp Key
CtrlShiftPlus EQUATE(036BH) !Ctrl-Shift-Plus Key
CtrlShiftPrint EQUATE(032CH) !Ctrl-Shift-PrintScreen Key
CtrlShiftRight EQUATE(0327H) !Ctrl-Shift-Cursor Right Key
CtrlShiftSlash EQUATE(036FH) !Ctrl-Shift-Slash Key
CtrlShiftSpace EQUATE(0320H) !Ctrl-Shift-Spacebar
CtrlShiftTab EQUATE(0309H) !Ctrl-Shift-Tab Key
CtrlShiftUp EQUATE(0326H) !Ctrl-Shift-Cursor Up Key

CtrlShiftPad0 EQUATE(0360H) !Ctrl-Shift-0 on numeric keypad
CtrlShiftPad1 EQUATE(0361H) !Ctrl-Shift-1 on numeric keypad
CtrlShiftPad2 EQUATE(0362H) !Ctrl-Shift-2 on numeric keypad
CtrlShiftPad3 EQUATE(0363H) !Ctrl-Shift-3 on numeric keypad
CtrlShiftPad4 EQUATE(0364H) !Ctrl-Shift-4 on numeric keypad
CtrlShiftPad5 EQUATE(0365H) !Ctrl-Shift-5 on numeric keypad
CtrlShiftPad6 EQUATE(0366H) !Ctrl-Shift-6 on numeric keypad
CtrlShiftPad7 EQUATE(0367H) !Ctrl-Shift-7 on numeric keypad
CtrlShiftPad8 EQUATE(0368H) !Ctrl-Shift-8 on numeric keypad
CtrlShiftPad9 EQUATE(0369H) !Ctrl-Shift-9 on numeric keypad


CtrlShiftMouseLeft EQUATE(0301H) !Ctrl-Shift-Left mouse button
CtrlShiftMouseRight EQUATE(0302H) !Ctrl-Shift-Right mouse button
CtrlShiftMouseCenter EQUATE(0303H) !Ctrl-Shift-Middle mouse button
CtrlShiftMouseLeft2 EQUATE(0305H) !Ctrl-Shift-Left mouse double-click
CtrlShiftMouseRight2 EQUATE(0306H) !Ctrl-Shift-Right mouse double-click
CtrlShiftMouseCenter2 EQUATE(0307H) !Ctrl-Shift-Middle mouse double-click

CtrlAlt0 EQUATE(0630H) !Ctrl-Alt-0 Key
CtrlAlt1 EQUATE(0631H) !Ctrl-Alt-1 Key
CtrlAlt2 EQUATE(0632H) !Ctrl-Alt-2 Key
```

```
CtrlAlt3 EQUATE(0633H) !Ctrl-Alt-3 Key
CtrlAlt4 EQUATE(0634H) !Ctrl-Alt-4 Key
CtrlAlt5 EQUATE(0635H) !Ctrl-Alt-5 Key
CtrlAlt6 EQUATE(0636H) !Ctrl-Alt-6 Key
CtrlAlt7 EQUATE(0637H) !Ctrl-Alt-7 Key
CtrlAlt8 EQUATE(0638H) !Ctrl-Alt-8 Key
CtrlAlt9 EQUATE(0639H) !Ctrl-Alt-9 Key

CtrlAltA EQUATE(0641H) !Ctrl-Alt-A Key
CtrlAltB EQUATE(0642H) !Ctrl-Alt-B Key
CtrlAltC EQUATE(0643H) !Ctrl-Alt-C Key
CtrlAltD EQUATE(0644H) !Ctrl-Alt-D Key
CtrlAltE EQUATE(0645H) !Ctrl-Alt-E Key
CtrlAltF EQUATE(0646H) !Ctrl-Alt-F Key
CtrlAltG EQUATE(0647H) !Ctrl-Alt-G Key
CtrlAltH EQUATE(0648H) !Ctrl-Alt-H Key
CtrlAltI EQUATE(0649H) !Ctrl-Alt-I Key
CtrlAltJ EQUATE(064AH) !Ctrl-Alt-J Key
CtrlAltK EQUATE(064BH) !Ctrl-Alt-K Key
CtrlAltL EQUATE(064CH) !Ctrl-Alt-L Key
CtrlAltM EQUATE(064DH) !Ctrl-Alt-M Key
CtrlAltN EQUATE(064EH) !Ctrl-Alt-N Key
CtrlAltO EQUATE(064FH) !Ctrl-Alt-O Key
CtrlAltP EQUATE(0650H) !Ctrl-Alt-P Key
CtrlAltQ EQUATE(0651H) !Ctrl-Alt-Q Key
CtrlAltR EQUATE(0652H) !Ctrl-Alt-R Key
CtrlAltS EQUATE(0653H) !Ctrl-Alt-S Key
CtrlAltT EQUATE(0654H) !Ctrl-Alt-T Key
CtrlAltU EQUATE(0655H) !Ctrl-Alt-U Key
CtrlAltV EQUATE(0656H) !Ctrl-Alt-V Key
CtrlAltW EQUATE(0657H) !Ctrl-Alt-W Key
CtrlAltX EQUATE(0658H) !Ctrl-Alt-X Key
CtrlAltY EQUATE(0659H) !Ctrl-Alt-Y Key
CtrlAltZ EQUATE(065AH) !Ctrl-Alt-Z Key

CtrlAltF1 EQUATE(0670H) !Ctrl-Alt-F1 Key
CtrlAltF2 EQUATE(0671H) !Ctrl-Alt-F2 Key
CtrlAltF3 EQUATE(0672H) !Ctrl-Alt-F3 Key
CtrlAltF4 EQUATE(0673H) !Ctrl-Alt-F4 Key
CtrlAltF5 EQUATE(0674H) !Ctrl-Alt-F5 Key
CtrlAltF6 EQUATE(0675H) !Ctrl-Alt-F6 Key
```

```
CtrlAltF7 EQUATE(0676H) !Ctrl-Alt-F7 Key
CtrlAltF8 EQUATE(0677H) !Ctrl-Alt-F8 Key
CtrlAltF9 EQUATE(0678H) !Ctrl-Alt-F9 Key
CtrlAltF10 EQUATE(0679H) !Ctrl-Alt-F10 Key
CtrlAltF11 EQUATE(067AH) !Ctrl-Alt-F11 Key
CtrlAltF12 EQUATE(067BH) !Ctrl-Alt-F12 Key

CtrlAltAst EQUATE(066AH) !Ctrl-Alt-Asterisk Key
CtrlAltBS EQUATE(0608H) !Ctrl-Alt-Backspace Key
CtrlAltDecimal EQUATE(066EH) !Ctrl-Alt-Decimal Key
CtrlAltDelete EQUATE(062EH) !Ctrl-Alt-Delete Key
CtrlAltDown EQUATE(0628H) !Ctrl-Alt-Cursor Down Key
CtrlAltEnd EQUATE(0623H) !Ctrl-Alt-End Key
CtrlAltEnter EQUATE(060DH) !Ctrl-Alt-Enter Key
CtrlAltEsc EQUATE(061BH) !Ctrl-Alt-Esc Key
CtrlAltHome EQUATE(0624H) !Ctrl-Alt-Home Key
CtrlAltInsert EQUATE(062DH) !Ctrl-Alt-Insert Key
CtrlAltLeft EQUATE(0625H) !Ctrl-Alt-Cursor Left Key
CtrlAltMinus EQUATE(066DH) !Ctrl-Alt-Minus Key
CtrlAltPause EQUATE(0613H) !Ctrl-Alt-Pause Key
CtrlAltPgDn EQUATE(0622H) !Ctrl-Alt-PgDn Key
CtrlAltPgUp EQUATE(0621H) !Ctrl-Alt-PgUp Key
CtrlAltPlus EQUATE(066BH) !Ctrl-Alt-Plus Key
CtrlAltPrint EQUATE(062CH) !Ctrl-Alt-PrintScreen Key
CtrlAltRight EQUATE(0627H) !Ctrl-Alt-Cursor Right Key
CtrlAltSlash EQUATE(066FH) !Ctrl-Alt-Slash Key
CtrlAltSpace EQUATE(0620H) !Ctrl-Alt-Spacebar
CtrlAltTab EQUATE(0609H) !Ctrl-Alt-Tab Key
CtrlAltUp EQUATE(0626H) !Ctrl-Alt-Cursor Up Key

CtrlAltPad0 EQUATE(0660H) !Ctrl-Alt-0 on numeric keypad
CtrlAltPad1 EQUATE(0661H) !Ctrl-Alt-1 on numeric keypad
CtrlAltPad2 EQUATE(0662H) !Ctrl-Alt-2 on numeric keypad
CtrlAltPad3 EQUATE(0663H) !Ctrl-Alt-3 on numeric keypad
CtrlAltPad4 EQUATE(0664H) !Ctrl-Alt-4 on numeric keypad
CtrlAltPad5 EQUATE(0665H) !Ctrl-Alt-5 on numeric keypad
CtrlAltPad6 EQUATE(0666H) !Ctrl-Alt-6 on numeric keypad
CtrlAltPad7 EQUATE(0667H) !Ctrl-Alt-7 on numeric keypad
CtrlAltPad8 EQUATE(0668H) !Ctrl-Alt-8 on numeric keypad
CtrlAltPad9 EQUATE(0669H) !Ctrl-Alt-9 on numeric keypad
```

```
CtrlAltMouseLeft EQUATE(0601H) !Ctrl-Alt-Left mouse button
CtrlAltMouseRight EQUATE(0602H) !Ctrl-Alt-Right mouse button
CtrlAltMouseCenter EQUATE(0603H) !Ctrl-Alt-Middle mouse button
CtrlAltMouseLeft2 EQUATE(0605H) !Ctrl-Alt-Left mouse double-click
CtrlAltMouseRight2 EQUATE(0606H) !Ctrl-Alt-Right mouse double-click
CtrlAltMouseCenter2 EQUATE(0607H) !Ctrl-Alt-Middle mouse double-click
```

Dictionary Class

The new Clarion threading model dictates that the existing File and Relation Managers use threaded objects (i.e. a new instance on every thread).

One of the effects of this is that the traditional ABC code that initializes both File and Relation Managers (contained in the DctInit generated procedure) now has to be executed whenever a new thread is started. Likewise, the Managers' kill code (traditionally contained in DctKill) must be called whenever a thread is terminated.

To facilitate this, a small globally defined class called **Dictionary** will be generated into every ABC template based application that does not have its global data defined external to the application. (i.e. the File and Relation managers compiled locally). The Dictionary object contains only construct and destruct methods but, more important, it is a *threaded object*.

Example:

```
Dictionary  CLASS,THREAD
Construct    PROCEDURE
Destruct     PROCEDURE
            END

Dictionary.Construct PROCEDURE
 CODE
 DctInit()

Dictionary.Destruct PROCEDURE
  CODE
  DctKill()
```

This means that the Construct method will be called whenever a new thread comes into existence and the Destruct method will be called whenever a thread is terminated. The constructor calls DctInit and the destructor calls DctKill. Therefore, DctInit is called whenever a thread is started and DctKill is now called whenever a thread is terminated; thus ensuring that threaded File and Relation managers are created and destroyed correctly.

FileCallBackInterface and DriverOps

The FileCallBackInterface methods are called before and after each file operation. For every FileCallBackInterface registered with a FILE or VIEW the **FileCallBackInterface.FunctionCalled** method is called before the operation (opcode) is performed. If any of these methods return FALSE, then the operation is not executed. After the operation has been called, or would have been called if **FileCallBackInterface.FunctionCalled** did not return FALSE, **FileCallBackInterface.FunctionDone** is called for every registered interface (See CALLBACK).

If either **FileCallBackInterface.FunctionCalled** or **FileCallBackInterface.FunctionDone** returns FALSE:

1. ERRORCODE() will be set to TRIGGERERROR (100)
2. FILEERRORCODE() will be set to the **FileErrCode** parameter of the first method that returned FALSE
3. FILEERROR() will be set to the **FileErrMsg** parameter of the first method that returned FALSE.

A **CSTRING(81)** is passed in FileErrCode and a **CSTRING(256)** is passed in FileErrMsg.

Both of the FileCallBackInterface methods receive an **opCode** and **Parameters**. Parameters are set up to offer different values depending on the particular opCode passed to the method.

The EQUATES.CLW file found in LIBSRC defines valid opCodes using the following format:

DriverOp:*Action*

Action	Invoking Statement	Valid Parameters
SEND	SEND(file, str)	Params.Text = str
COPY	COPY(file, str)	Params.Text = str
RENAME	RENAME(file,str)	Params.Text = str
SETNULLS	SETNULLS(file,str)	Params.Text = str

SETProperty	file{property,index} = value	Params.Text = value
		Params.Property =property
		Params.Index = index

SETBlobProperty	blob{property} = value	Params.Index = blob number in file definition (1 based) Params.Text = value Params.Property = property
SETKeyProperty	key{property,index} = value	Params.Text = value Params.Property = property Params.Key1 = key Params.Index = index
DOProperty	file{property, index}	
GetProperty	x = file{property,index}	
DOBlobProperty	blob{property}	Params.Index =blob number in file definition (1 based) Params.Property = property
GetBlobProperty	x = blob{property}	Params.Index = blob number in file definition (1 based) Params.Property = property
DOKeyProperty	Key{property, index}	Params.Property = property Params.Key1 = key Params.Index = index
GetKeyProperty	x = key{property}	Params.Property = property Params.Key1 = key Params.Index = index
BUILDdyn	BUILD(INDEX, field_list)	Params.FieldList = field_list Params.Key1 = INDEX
BUILDdynfilter	BUILD(INDEX,	Params.FieldList = field_list Params.Text = filter

	field_list, filter)	Params.Key1 = INDEX
REGETfile	REGET(file,buffer)	Params.Position = buffer
RESETfile	RESET(file,buffer)	Params.Position = buffer
REGETkey	REGET(key,buffer)	Params.Position = buffer Params.Key1 = key
RESETkey	RESET(key,buffer)	Params.Position = buffer Params.Key1 = key
ADDfilelen	ADD(file,length)	Params.Len = length
APPENDlen	APPEND(file,length)	Params.Len = length
BUFFER	BUFFER(file,pagesize, behind, ahead, timeout)	Params.Buffer = pagesize Params.Behind = behind Params.Ahead = ahead Params.TimeOut = timeout
BUILDkey	BUILD(key)	Params.Key1 = key
DUPLICATEkey	DUPLICATE(key)	Params.Key1 = key
GETfilekey	GET(file,key)	Params.Key1 = key
POINTERkey	x = POINTER(key)	Params.Key1 = key

POSITIONkey	x = POSITION(key)	Params.Key1 = key
RECORDSkey	x = RECORDS(key)	Params.Key1 = key
SETfilekey	SET(file,key)	Params.Key1 = key
SETkey	SET(key)	Params.Key1 = key
OPEN	OPEN(file, mode)	Params.OpenMode = mode
SHARE	SHARE(file, mode)	Params.OpenMode = mode
GETfileptrlen	GET(file, pointer, length)	Params.Pointer = pointer Params.Len = length
PUTfileptrlen	PUT(file, pointer, length)	Params.Pointer = pointer Params.Len = length
GETfileptr	GET(file, pointer)	Params.Pointer = pointer
PUTfileptr	PUT(file, pointer)	Params.Pointer = pointer
SETfileptr	SET(file, pointer)	Params.Pointer = pointer
GETkeyptr	GET(key, pointer)	Params.Key1 = key Params.Pointer = pointer

SETkeyptr	SET(key, pointer)	Params.Key1 = key Params.Pointer = pointer
HOLDfilesec	HOLD(file, seconds)	Params.Seconds = seconds
LOCKfilesec	LOCK(file, seconds)	Params.Seconds = seconds
SETkeykeyptr	SET(key1, key2, pointer)	Params.Key1 = key1 Params.Key1 = key2 Params.Pointer = pointer
SETkeykey	SET(key1, key2)	Params.Key1 = key1 Params.Key1 = key2
SKIP	SKIP(file, records)	Params.Records = records
NULL	x = NULL(field)	Params.Field = ADDRESS(field) Params.Len = SIZE(field)
SETNULL	SETNULL(field)	Params.Field = ADDRESS(field) Params.Len = SIZE(field)
SETNONNULL	SETNONULL(field)	Params.Field = ADDRESS(field) Params.Len = SIZE(field)
CLEARfile	CLEAR(file, n)	Params.Index = n
DESTROY	called on termination of application or thread	Params.Index = THREAD()
GETblobdata	x = blob[s : e]	Params.Index =blob number in file definition (1 based) Params.Start = s Params.Stop = e

SETblobdata	blob[s : e] = x	Params.Index = blob number in file definition (1 based) Params.Start = s Params.Stop = e
BLOBSIZE	x = SIZE(blob)	Params.Index =blob number in file definition (1 based)
SETviewfields	SET(view, fields = 0)	Params.Fields = fields
RESETviewf	RESET(view, file)	Params.File = file
RESTORESTATE	RESTORESTATE(file, state, restorebuffer)	Params.State = state Params.RestoreBuffer = restorebuffer
FREESTATE	FREESTATE(file, state)	Params.State = state
VIEWSTART	internal operation	Params.TransactionOpCode = operation starting or stopping
VIEWEND	internal operation	Params.TransactionOpCode = operation starting or stopping

Note:

VIEWSTART and VIEWEND are called by the ISAM View Engine when it executes NEXT, PREV, POSITIONfile, REGETfile, RESETfile, GETfileptr, RECORDSfile,and SKIP.

STARTTRAN	internal operation	Params.TransactionOpCode = operation starting or stopping
ENDTRAN	internal operation	Params.TransactionOpCode = operation starting or stopping

Note:

The runtime library will call the file driver once when starting a transaction statement (LOGOUT, COMMIT, or ROLLBACK) with an opCode of **DriverOps:STARTTRAN** and once when ending the transaction statement with an opCode of **DriverOps:ENDTRAN**. If a callback has been registered with the first file in the logout file list, then it will be called with opCode set to **DriverOps:STARTTRAN** at the start of the transaction command and **DriverOps:ENDTRAN** at the end of the transaction command. In this case Parameters.TransactionOpCode will be set to either **DriverOps:LOGOUT**, **DriverOps:COMMIT** or **DriverOps:ROLLBACK**.

API Calls and Advanced Resources

Prototypes and Declarations

Clarion includes files with prototypes, declarations, and headers that you can use to let Clarion "talk" to Windows, C/C++, Modula-2, and vice versa.

Clarion to C/C++ Standard Library

To call the standard C library functions from Clarion applications, include \CLIB.CLW in the "Inside the Global Map" embed point.

```
INCLUDE('CLIB.CLW')
```

This file contains Clarion prototypes for various string handling functions, integer math, character type functions, and low level file manipulation functions. Refer to your C/C++ Library Reference for more information on individual functions.

Clarion to Windows API

To call Windows API functions from Clarion applications, you must include the functions' prototypes in your application's MAP structure, and any standard EQUATEs or data structures that the functions need in your Global data declarations.

Clarion contains the WINAPI.EXE utility program that creates the file you need to include in your application. This program, by default, creates the WINAPI.CLW file which has two sections: the "Equates" section containing all EQUATE statements and any data structures needed by the functions you choose, and the "Prototypes" section containing the Clarion language prototypes of Windows API functions you choose to use.

Include the Equates section of WINAPI.CLW in the "After Global INCLUDEs" embed point:

```
INCLUDE('WINAPI.CLW','Equates')
```

Include the Prototypes section of WINAPI.CLW in the "Inside the Global Map" embed point:

```
INCLUDE('WINAPI.CLW','Prototypes')
```

Refer to your Windows API reference for more information on the individual API functions available to you in categories such as:

Creating Windows
Window Support
Message Processing
Memory Management
Bitmaps and Icons
Color Palette Control
Sound
Character Sets and Strings
Communications
Metafiles
Tool Help Library
File Compression

Installation and Version Information
TrueType Fonts
Multimedia

Modula-2 to Clarion

Clarion's Runtime Library

To call the Clarion runtime library procedures create a definition file (.DEF) that contains Modula-2 declarations for various Clarion Language procedures, as well as the many standard C library functions that are found in the Clarion Runtime Library. The available functions are documented in the Clarion's Runtime Library Functions section of this section.

Clarion's File Driver Procedures

To call the Clarion database file driver procedures create a definition file (.DEF) that contains Modula-2 declarations for Clarion's FILE, RECORD, KEY, INDEX, MEMO, and BLOB handling procedures, including a complete description of Clarion's file control block.

C/C++ to Clarion

Clarion's Runtime Library

To call the Clarion runtime library procedures create a header file (.H) that contains C/C++ prototypes for various Clarion Language procedures, as well as many standard C library functions that are found in the Clarion Runtime Library. The available functions are documented below in the Clarion's Runtime Library Functions section.

Clarion's File Driver Procedures

To call the Clarion database file driver procedures create a header file (.H) that contains contains C/C++ prototypes for Clarion's FILE, RECORD, KEY, INDEX, MEMO, and BLOB handling procedures, including a complete description of Clarion's file control block.

Accessing Clarion's Runtime Library from C/C++ or Modula-2 Code

Following is a list of Clarion runtime library procedures, data structures, and variables that you may use at run time in your C/C++ or Modula-2 code.

Structures and Data Type Definitions

COLORREF
C++: typedef unsigned long COLORREF;
Modula-2: TYPE COLORREF = LONGINT;

Run-Time Variables

The following variables are available for interrogation at run-time:

Cla$DOSerror An unsigned integer containing the last DOS error code.

Cla$FILEERRCODE An integer containing the last Clarion error code.

Cla$FILEERRORMSG A character array of 80 char's containing the last Clarion error message.

WSL@AppInstance An unsigned short containing the instance ID of the application.

Clarion Built-in Procedures

The following list of procedures are those internal Clarion procedures that are 'safe' to call at run-time. Unless otherwise stated, assume that these procedures have been given external C linkage.

Cla$ACOS The Clarion ACOS() procedure. Returns the inverse cosine of the val parameter.
C++: double Cla$ACOS(double val)
Modula-2: Cla$ACOS(val :LONGREAL):LONGREAL;

val: A numeric expression describing an angle in radians.

Cla$ARC The Clarion ARC statement. Places an arc of an ellipse on the current window or report, bounded by the rectangle defined by the x, y, wd and ht parameters.

C++: void Cla$ARC(int x, int y, int wd, int ht, int start, int end)
Modula-2: Cla$ARC(x,y,wd,ht,start,end: INTEGER);

x: An integer specifying the horizontal position of the starting point.

y: An integer specifying the vertical position of the starting point.

wd: An integer specifying then width.

ht: An integer specifying then height.

start: An integer specifying the start of the arc in 10th's of a degree.

end: An integer specifying the end of the arc in 10th's of a degree.

Cla$ASIN The Clarion ASIN() procedure. Returns the inverse sine of the val parameter.

C++: double Cla$ASIN(double val)

Modula-2: Cla$ASIN(val LONGREAL): LONGREAL;

val: A numeric expression describing an angle in radians.

Cla$ATAN The Clarion ATAN() procedure. Returns the inverse tangent of the val parameter.

C++: double Cla$ATAN(double val)

Modula-2: Cla$ATAN(val: LONGREAL):LONGREAL;

val: A numeric expression describing an angle in radians.

Cla$BOX The Clarion BOX statement. This procedure draws a box of the color specified by the COLORREF structure, starting at position x, y of the width and height specified on the current window or report.

C++: void Cla$BOX(int x, int y, int wd, int ht, COLORREF fillcolor)

Modula-2: Cla$BOX(x, y, wd, ht: INTEGER; fillcolor: COLORREF);

x: An integer specifying the horizontal start position.

y: An integer specifying the vertical start position.

wd: An integer specifying the width.

ht: An integer specifying the height.

fillcolor: A COLORREF structure.

Cla$BSHIFT The Clarion BSHIFT() procedure. This procedure returns the result of bit shifting val by count binary positions. If count is positive, val is shifted left, if count is negative val is shifted right.

C++: long Cla$BSHIFT(long val, int count)

Modula-2: Cla$BSHIFT(val: LONGINT; count: INTEGER): LONGINT;

val: A numeric expression.

count: A numeric expression.

Cla$CHORD The Clarion CHORD statement. Draws a closed sector ellipse on the current window or report inside the box specified by the x, y, wd and ht parameters and in the color provided in the COLORREF structure. The start and end parameters specify which part of the ellipse to draw.

C++: void Cla$CHORD(int x, int y, int wd, int ht, int start, int end, COLORREF fillcolor)

Modula-2: Cla$CHORD(x, y, wd, ht, start, end: INTEGER; fillcolor: COLORREF);

x: An integer specifying the horizontal start position.

y: An integer specifying the vertical start position.

wd: An integer specifying the width.

ht: An integer specifying the height.

start: An integer expressing the string of the chord in 10th’s of a degree.

end: An integer expressing the end of the chord in 10th’s of a degree.

fillcolor: A COLORREF structure.

Cla$CLOCK The Clarion CLOCK() procedure. Returns the system time in the form of a Clarion standard time.

C++: long Cla$CLOCK(void)

Modula-2: Cla$CLOCK(): LONGINT;

Cla$COS The Clarion COS() procedure. Returns the cosine of the val parameter.

C++: double Cla$COS(double val)

Modula-2: Cla$COS(val: LONGREAL): LONGREAL;

val: A numeric expression describing an angle in radians.

Cla$DATE The Clarion DATE() procedure. Returns a Clarion standard date value form the component day, month and year parameters.

C++: long Cla$DATE(unsigned mn, unsigned dy, unsigned yr)

Modula-2: Cla$DATE(mn, dy, yr: CARDINAL): LONGINT;

mn: A numeric expression for the month in the range 1 to 12.

dy: A numeric expression for the day in the range 1 to 31.

yr A numeric expression for the year in the range 1801 to 2099.

Cla$DAY The Clarion DAY() procedure. Returns the day in the range 1 to 31 from the Clarion standard date parameter.

C++: long Cla$DAY(long dt)

Modula-2: Cla$DAY(dt: LONGINT): LONGINT;

dt: A numeric expression for Clarion standard date.

Cla$ELLIPSE The Clarion ELLIPSE statement. Draws an ellipse on the current window or report, of the color specified in the COLORREF structure, inside the area bounded by the x, y, wd and ht parameters.

C++: void Cla$ELLIPSE(int x, int y, int wd, int ht, COLORREF fillcolor)

Modula-2: Cla$ELLIPSE(x, y, wd, ht: INTEGER; fillcolor: COLOREF);

x: An integer expression.

y: An integer expression.

wd: An integer expression.

ht: An integer expression.

fillcolor: A COLORREF structure.

Cla$INT The Clarion INT() procedure. Returns the integer portion of the val parameter. The value is truncated at the decimal point and no rounding is performed.

C++: double Cla$INT(double val)

Modula-2: Cla$INT(val: LONGREAL): LONGREAL;

val: A numeric expression.

Cla$LOG10 The Clarion LOG10() procedure. Returns the base 10 logarithm of the val parameter.

C++: double Cla$LOG10(double val)

Modula-2: Cla$LOG10(val: LONGREAL): LONGREAL;

val: A numeric expression.

Cla$LOGE The Clarion LOGE() procedure. Returns the natural logarithm of the val parameter.

C++: double Cla$LOGE(double val)

Modula-2: Cla$LOGE(val: LONGREAL): LONGREAL;

val: A numeric expression.

Cla$MONTH The Clarion MONTH() procedure. Returns the month from a Clarion standard date in the range 1 to 12.

C++: long Cla$MONTH(long dt)

Modula-2: Cla$MONTH(dt: LONGINT): LONGINT;

dt: A numeric expression containing a Clarion standard date.

Cla$MOUSEX The Clarion MOUSEX() procedure. Returns the horizontal position of the mouse.

C++: int Cla$MOUSEX(void)

Modula-2: Cla$MOUSEX(): INTEGER;

Cla$MOUSEY The Clarion MOUSEY() procedure. Returns the horizontal position of the mouse.

C++: int Cla$MOUSEY(void)

Modula-2: Cla$MOUSEY(): INTEGER;

Cla$NUMERIC The Clarion NUMERIC() procedure. Returns 1 (true) if str contains a valid representation of a number, otherwise returns 0 (false).

C++: unsigned Cla$NUMERIC(char *str, unsigned slen)

Modula-2: Cla$NUMERIC(VAR str: ARRAY OF CHAR; slen:CARDINAL): CARDINAL;

str: A pointer to a string.

slen: Length of the str parameter.

Cla$RANDOM The Clarion RANDOM() procedure. Returns a pseudo-random number who’s value will be between the low and high bound values.

C++: long Cla$RANDOM(long low, long high)

Modula-2: Cla$RANDOM(low, high: LONGINT): LONGINT;

low: A numeric value specifying the lower bound.

high: A numeric value specifying the upper bound.

Cla$ROUND The Clarion ROUND() procedure. Returns the val parameter rounded to power of 10 specified by the ord parameter.

C++: double Cla$ROUND(double val, double ord)

Modula-2: Cla$ROUND(val, ord: LONGREAL): LONGREAL;

val: A numeric expression.

ord: A numeric expression equal to a power of 10 (e.g. .001, .0, 1, 10, 100 etc...).

Cla$SETCLOCK
The Clarion SETCLOCK statement. Sets the system clock to the time contained in the dt parameter.

C++: void Cla$SETCLOCK(long dt)

Modula-2: Cla$SETCLOCK(dt: LONGINT);

dt: A numeric expression representing a Clarion standard time.

Cla$SETTODAY
The Clarion SETTODAY statement. Sets the DOS system date to that contained in the dt parameter.

C++: void Cla$SETTODAY(long dt)

Modula-2: Cla&SETTODAY(dt: LONGINT);

dt: A numeric expression containing a Clarion standard date.

Cla$SIN The Clarion SIN() procedure. Returns the sine of the val parameter.

C++: double Cla$SIN(double val)

Modula-2: CLA$SIN(val: LONGREAL): LONGREAL;

val: A numeric expression describing an angle in radians.

Cla$SQRT The Clarion SQRT() procedure. Returns the square root of the val parameter.

C++: double Cla$SQRT(double val)

Modula-2: Cla$SQRT(val:LONGREAL): LONGREAL;

val: A numeric expression.

Cla$TAN The Clarion TAN() procedure. Returns the tangent of the val parameter.

C++: double Cla$TAN(double val)

Modula-2: Cla$TAN(val: LONGREAL): LONGREAL;

val: A numeric expression describing an angle in radians.

Cla$TODAY The Clarion TODAY() procedure. Returns the system date in Clarion standard date format.

C++: long Cla$TODAY(void)

Modula-2: Cla$TODAY(): LONGINT;

Cla$YEAR The Clarion YEAR() procedure. Extracts the year from a Clarion standard date, in the range 1801 to 2099.

C++: long Cla$YEAR(long dt)

Modula-2: Cla$YEAR(dt: LONGINT): LONGINT;

dt: A numeric expression describing a Clarion standard date.

Clarion String Stack Handling Procedures

The following section describes the use Clarion internal run-time string handling procedures available to 3GL code. Clarion uses a LISP like approach to string handling whereby, parameters are pushed onto the top of the string stack, with operations being performed on the topmost entries. Assume, unless otherwise documented, that the procedures remove (or Pop) items off the stack that they have used.

Please note that some of the following procedures require pointers to null terminated strings, to be passed as parameters. Modula-2 programmers should use the Modula library procedure Str.StrToC to convert strings to null terminated equivalents. Also, the pragma call(o_a_size=>off,o_a_copy=>off) must be issued to prevent the passing of array size information to the run-time procedures.

Cla$PopCString Takes the topmost item off the stack and copies it to the string pointed to by s; len contains the length of the string copied to s.

C++: void Cla$PopCString(char *s, unsigned len)

Modula-2: Cla$PopCString(s: POINTER TO CHAR; len: CARDINAL);

s: A pointer to a null terminated string

len: The length of string s

Cla$PopPString Takes the topmost item off the stack and copies it to the string pointed to by s; len contains the length of the string copied to s. The string is converted to a Pascal style string (i.e. first byte is string length) during copy.

C++: void Cla$PopPString(char *s, unsigned len)

Modula-2: Cla$PopPString(VAR s: ARRAY OF CHAR; len: CARDINAL);

s: A pointer to a string

len: The length of string s

Cla$PopString Pops the uppermost stack item and copies it to the string s.

C++: void Cla$PopString(char *s, unsigned len)

Modula-2: Cla$PopString(VAR s: ARRAY OF CHAR; len: CARDINAL);

s: A pointer to a null terminated string

len: The length of string s

Cla$PushCString Pushes s onto the top of the stack.

C++: void Cla$PushCString(char *s)

Modula-2: Cla$PushCString(VAR s: ARRAY OF CHAR);

s: A pointer to a null terminated string

Cla$PushString Pushes the string s onto the top of the stack. Len specifies the length of string s.

C++: void Cla$PushString(char *s, unsigned len)

Modula-2: Cla$PushString(VAR s: ARRAY OF CHAR; len: CARDINAL);

s: A pointer to a string

len: The length of string s

Cla$StackALL The Clarion ALL() procedure. Pops the top item of the stack and replaces it by a string containing the original string replicated as many times as necessary to produce a string of length len.

C++: void Cla$StackALL(unsigned len)

Modula-2: Cla$StackALL(len: CARDINAL);

len: An unsigned integer

Cla$StackCENTER The Clarion CENTER() procedure. Pops the topmost item of the stack and replaces it with a string padded with leading spaces so as to center the text in a string of length len.

C++: void Cla$StackCENTER(unsigned len)

Modula-2: Cla$StackCENTER(len: CARDINAL);

len: An unsigned integer

Cla$StackCLIP The Clarion CLIP() procedure. Removes trailing spaces from the top most item on the stack.

C++: void Cla$StackCLIP(void)

Modula-2: Cla$StackCLIP();

Cla$StackCompare Compares the top item on the stack (s1) with the 2nd item on the stack (s2) and returns one of the following values:

-1: if s1 < s2

0: if s1 = s2

1: if s1 > s2

After the compare instruction, s1 and s2 are removed from the stack automatically.

C++: int Cla$StackCompare(void)

Modula-2: Cla$StackCompare(): INTEGER;

Cla$StackCompareN Compares the topmost item on the stack to null. Returns true if the topmost item is null, otherwise returns false.

C++: int Cla$StackCompareN(void)

Modula-2: Cla$StackCompareN(): INTEGER;

Cla$StackConcat Pops the top two items off the stack, concatenates them together and pushes the resulting string back onto the stack.

C++: void Cla$StackConcat(void)

Modula-2: Cla$StackConcat();

Cla$StackINSTRING The Clarion INSTRING() procedure. Searches the topmost item on the stack, for any occurrence of the second item on the stack. The search starts at character position start and increments the start position by step until the end of the string is reach. Returns the iteration count required to find the search string, or 0 if not found.

C++: unsigned Cla$StackINSTRING(unsigned step, unsigned start)

Modula-2: Cla$StackINSTRING(step, start: CARDINAL): CARDINAL;

step: An unsigned integer, the search increment

start: An unsigned integer, the start position of the search

Cla$StackLEFT The Clarion LEFT() procedure. Replaces the topmost string on the stack with its left justified equivalent. The replacement sting will have a length of len.

C++: void Cla$StackLEFT(unsigned len)

Modula-2: Cla$StackLEFT(len: CARDINAL);

len: An unsigned integer

Cla$StackLen Returns the length of the topmost item on the stack. Does not pop the item off the stack.

C++: unsigned Cla$StackLen(void)

Modula-2: Cla$StackLen(): CARDINAL;

Cla$StackLen2 Returns the length of the topmost item on the stack. Pops the item of the stack after getting its length.

C++: unsigned Cla$StackLen2(void)

Modula-2: Cla$StackLen2(): CARDINAL;

Cla$StackLOWER The Clarion LOWER() procedure. Replaces the topmost string on the stack with its lower case equivalent.

C++ void Cla$StackLOWER(void)

Modula-2: Cla$StackLOWER();

Cla$STACKpop Pops the top item off the stack.

C++: void Cla$STACKpop(void)

Modula-2: Cla$STACKpop();

Cla$StackNUMERIC Returns true if the topmost string on stack contains a valid numeric representation, otherwise returns false.

C++: unsigned Cla$StackNUMERIC(void)

Modula-2: Cla$StackNUMERIC(): CARDINAL;

Cla$StackPRESS The Clarion PRESS statement. Pushes every character in the topmost string of the stack into the Windows keyboard buffer.

C++: void Cla$StackPRESS(void)

Modula-2: Cla$StackPRESS();

Cla$StackRIGHT The Clarion RIGHT() procedure. Replaces the topmost item on the stack with its right justified equivalent. The replacement string will have a length of len characters.

C++: void Cla$StackRIGHT(unsigned len)

Modula-2: Cla$StackRIGHT(len: CARDINAL);

len: An unsigned integer

Cla$StackSUB The Clarion SUB() procedure. Replaces the topmost string on the stack with a sub slice of the string starting at character position pos and of length len.

C++: void Cla$StackSUB(unsigned pos, unsigned len)

Modula-2: Cla$StackSUB(pos, len: CARDINAL);

pos: An unsigned integer; the start position of the sub string

len: An unsigned integer; the length of the sub string

Cla$StackVAL The Clarion VAL() procedure. Returns the ANSI value of the first character of the topmost string of the stack.

C++: unsigned char Cla$StackVAL(void)

Modula-2: Cla$StackVAL(): BYTE;

Cla$StackUPPER Replace the topmost string on the stack with its uppercase equivalent.

C++: void Cla$StackUPPER(void)

Modula-2: Cla$StackUPPER();

Standard C Functions in Clarion's Runtime Library

Conversion Functions

Integer Math

Char Type Functions

Utility Functions

String Functions

Low-Level File Manipulation

The following functions comprise a sub-set of the standard SoftVelocity library that you can call from your Clarion, C/C++, or Modula-2 code. All of these functions are fully documented in the SoftVelocity C Library Reference manual (or in any ANSI-standard C library reference) and so, are not documented here. Unless otherwise indicated, assume that the functions operate exactly as documented.

The purpose of this list is simply to let you know what C standard library functions are available and the correct prototypes for each language.

Conversion Functions

Please note that some of the following functions require pointers to null terminated strings as parameters. Modula-2 programmers should use the Modula library procedure Str.StrToC to convert strings to null terminated equivalents. Also, the pragma call(o_a_size=>off, o_a_copy=>off) must be issued to prevent the passing of array size information to the run-time procedures.

atof Convert string to floating point.

C++: double atof(const char *_nptr)

Modula-2: atof(VAR _nptr: ARRAY OF CHAR): LONGREAL;

Clarion: AToF(*cstring),real,raw,name('_atof')

atoi Convert string to integer.

C++: int atoi(const char *_nptr)

Modula-2: atoi(VAR _nptr: ARRAY OF CHAR): INTEGER;

Clarion: AToI(*cstring),short,raw,name('_atoi')

atol Convert string to long.

C++: long atol(const char *_nptr)

Modula-2: atol(VAR _nptr: ARRAY OF CHAR): LONGINT;

Clarion: AToL(*cstring),long,raw,name('_atol')

atoul Convert string to unsigned long.

C++: unsigned long atoul(const char *_nptr)

Modula-2: atoul(VAR _nptr: ARRAY OF CHAR): LONGCARD;

Clarion: AToUL(*cstring),ulong,raw,name('_atoul')

Integer Math

abs Integer absolute value.

C++: int abs(int _num)

Modula-2: abs(_num: INTEGER): INTEGER;

Clarion: API_Abs(short),short,name('_abs') !Renamed to avoid conflict with Builtins.Clw

labs Long integer absolute value.

C++: long labs(long _j)

Modula-2: labs(_i: LONGINT): LONGINT;

Clarion: LAbs(long),long,name('_labs')

Char Type Functions

The following functions have only been tested when implemented as functions. We do not advise defining _CT_MTF to implement the functions as macros.

toupper Test and convert if lowercase.

C++: int toupper(int c)

Modula-2: toupper(c: INTEGER):INTEGER;

Clarion: ToUpper(short),short,name('_toupper')

Tolower Test and convert if uppercase.

C++: int tolower(int c)

Modula-2: tolower(c: INTEGER): INTEGER;

Clarion: ToLower(short),short,name('_tolower')

isascii ASCII test function.

C++: int isascii(int c)

Modula-2: isascii(c: INTEGER): INTEGER;

Clarion: IsAscii(short),short,name('_isascii')

iscntrl Control character test function.

C++: int iscntrl(int c)

Modula-2: iscntrl(c: INTEGER): INTEGER;

Clarion: IsCntrl(short),short,name('_iscntrl')

isdigit Numerics test function.

C++: int isdigit(int c)

Modula-2: isdigit(c: INTEGER): INTEGER;

Clarion: IsDigit(short),short,name('_isdigit')

Isprint Printable including space test function.

C++: int isprint(int c)

Modula-2: isprint(c: INTEGER): INTEGER;

Clarion: IsPrint(short),short,name('_isprint')

Ispunct Punctuation character test function.

C++: int ispunc(int c)

Modula-2: ispunc(c: INTEGER): INTEGER;

Clarion: IsPunct(short),short,name('_ispunct')

isspace Whitespace test function.

C++: int isspace(int c)

Modula-2: isspace(c: INTEGER): INTEGER;

Clarion: IsSpace(short),short,name('_isspace')

Isxdigit Hex digit test function.

C++: int isxdigit(int c)

Modula-2: isxdigit(c: INTEGER): INTEGER;

Clarion: IsXDigit(short),short,name('_isxdigit')

Utility Functions

rand Return pseudorandom integer.

C++: int rand(void)

Modula-2: rand(): INTEGER;

Clarion: Rand(),short,name('_rand')

randomize Set pseudorandom seed with system time.

C++: void randomize(void)

Modula-2: randomize()

Clarion: Randomize(),name('_randomize')

srand Set pseudorandom seed with specified number.

C++: void srand(unsigned _seed)

Modula-2: srand(_seed: CARDINAL);

Clarion: SRand(ushort),name('_srand')

String Functions

strcat Concatenate two strings.

C++: char *strcat(char *_dest, const char *_source)

Modula-2: Not available

Clarion: StrCat(*cstring,*cstring),cstring,raw,name('_strcat')

strcmp Compare two strings.

C++: int strcmp(const char *_s1, const char *_s2)

Modula-2: Not available

Clarion: StrCmp(*cstring,*cstring),short,raw,name('_strcmp')

chrcmp Compare two characters

C++: int chrcmp(char _c1, char _c2)

Modula-2: chrcmp(_c1,_c2: CHAR): INTEGER;

Clarion: ChrCmp(byte,byte),short,name('_chrcmp')

strequ

C++: int strequ(const char *_s1, const char *_s2)

Modula-2: Not available

Clarion: StrEqu(*cstring,*cstring),short,raw,name('_strequ')

strcpy Copy one string to another, return destination address.

C++: char *strcpy(char *_dest, const char *_source)

Modula-2: Not available

Clarion: StrCpy(*cstring, *cstring), cstring, raw,| name('_strcpy')

strlen Return string length.

C++: unsigned strlen(const char *_s)

Modula-2: strlen(VAR _s: ARRAY OF CHAR): CARDINAL;

Clarion: StrLen(*cstring),ushort,raw,name('_strlen')

strchr Find character in string.

C++: char *strchr(const char *_s, int _c)

Modula-2: Not available

Clarion: StrChr(*cstring,short),cstring,raw,name('_strchr')

strcspn Finds one of a set of characters in string.

C++: unsigned strcspn(const char *_s1, const char *_s2)

Modula-2: Not available

Clarion: StrCSpn(*cstring, *cstring), ushort, raw,| name('_strcspn')

strspn Find first character with no match in given character set.

C++: unsigned strspn(const char *_s1, const char *_s2)

Modula-2: Not available

Clarion: StrSpn(*cstring,*cstring),ushort,raw,name('_strspn')

strstr Find first occurrence of substring in a string.

C++: char *strstr(const char *_s1, const char *_s2)

Modula-2: Not available

Clarion: StrStr(*cstring,*cstring),cstring,raw,name('_strstr')

strtok Find next token in string.

C++: char *strtok(char *_s1, const char *_s2)

Modula-2: Not available

Clarion: StrTok(*cstring,*cstring),cstring,raw,name('_strtok')

strpbrk Find first occurrence of character.

C++: char *strpbrk(const char *_s1, const char *_s2)

Modula-2: Not available

Clarion: StrPBrk(*cstring, *cstring), cstring, raw,| name('_strpbrk')

strrchr Find last occurrence of character.

C++: char *strrchr(const char *_s, int _c)

Modula-2: Not available

Clarion: StrRChr(*cstring,short),cstring,raw,name('_strrchr')

strlwr Convert to lower case.

C++: char *strlwr(char *_s)

Modula-2: Not available

Clarion: StrLwr(*cstring),cstring,raw,name('_strlwr')

strupr Convert to upper case.

C++: char *strupr(char *_s)

Modula-2: Not available

Clarion: StrUpr(*cstring),cstring,raw,name('_strupr')

strdup Duplicate string.

C++: char *strdup(const char *_s)

Modula-2: Not available

Clarion: StrDup(*cstring),cstring,raw,name('_strdup')

strrev Reverse string.

C++: char *strrev(char *_s)

Modula-2: Not available

Clarion: StrRev(*cstring),cstring,raw,name('_strrev')

strncat Concatenate n characters.

C++: char *strncat(char *_dest, const char *_source, unsigned _n)

Modula-2: Not available

Clarion: StrNCat(*cstring, *cstring, ushort), cstring, raw,| name('_strncat')

strncmp Compare n characters.

C++: int strncmp(const char *_s1, const char *_s2, unsigned _n)

Modula-2: Not available

Clarion: StrNCmp(*cstring, *cstring, ushort), short, raw,| name('_strncmp')

strncpy Copy n characters.

C++: char * strncpy(char *_dest, const char *_source, unsigned _n)

Modula-2: Not available

Clarion: StrNCpy(*cstring, *cstring, ushort), cstring, raw,| name('_strncpy')

strnicmp Compare n characters regardless of case.

C++: int stricmp(const char *_s1, const char *_s2, unsigned _n)

Modula-2: Not available

Clarion: StrNICmp(*cstring, *cstring, ushort), short, raw,| name('_strnicmp')

Low-Level File Manipulation

access checks whether the file (or directory) specified by the path parameter exists, and (if not a directory) whether it can be accessed in the specified mode. The function returns 0 if access is permitted using the specified mode, -1 if it fails the check. Mode of 0 checks for existence, mode of 2 checks for write permission, mode of 4 checks for read permission, and 6 checks for read/write.

C++: int_access(const char *path,int access)

Modula-2: _access(VAR path: ARRAY OF CHAR; access:INTEGER): INTEGER;

Clarion: Access(*cstring,short),short,raw,name('_access')

chmod Set file's access mode.

C++: int _chmod(const char *path, int mode)

Modula-2: _chmod(VAR path: ARRAY OF CHAR; mode: INTEGER): INTEGER;

Clarion: ChMod(*cstring,short),short,raw,name('_chmod')

remove Deletes the file specified by the path parameter.

C++: int _remove(const char *_path)

Modula-2: _remove(VAR _path: ARRAY OF CHAR): INTEGER;

Clarion: API_Remove(*cstring),short,raw,name('_remove')

!Renamed to avoid conflict with Builtins.Clw

rename Changes the name of the file or directory specified by the oldname parameter.

C++: int _rename(const char *_oldname, const char *_newname)

Modula-2: _rename(VAR _oldname, VAR _newname: ARRAY OF CHAR): INTEGER;

Clarion: API_Rename(*cstring, *cstring), short, raw,| name('_rename')

!Renamed to avoid conflict with Builtins.Clw

fnmerge Builds a complete path name from its component parts -- drive, directory, filename, and extension.

C++: void _fnmerge(char *_path, const char *_drive, const char *_dir, const char *_name, const char *_ext)

Modula-2: _fnmerge(VAR _path, VAR _drive, VAR _dir, VAR _name, VAR _ext: ARRAY OF CHAR);

Clarion: FnMerge(*cstring, *cstring, *cstring, *cstring,| *cstring), raw, name('_fnmerge')

Note: fnmerge is actually called using PathMerge in the prototypes in CWUTIL.INC

fnsplit This function breaks a complete path name into its component parts -- drive, directory, filename, and extension.

C++: int _fnsplit(const char *_path, char *_drive, char *_dir, char *_name, char *_ext)

Modula-2: _fnsplit(VAR_path,VAR _drive,VAR _dir,VAR _name,VAR _ext:ARRAY OF CHAR):INTEGER;

Clarion: FnSplit(*cstring, *cstring, *cstring, *cstring,| *cstring), short, raw, name('_fnsplit')

Note: fnsplit is actually called using PathSplit using the prototypes in CWUTIL.INC.

mkdir Creates a new directory with the name passed in the path parameter.

C++: int _mkdir(const char *_path)

Modula-2: _mkdir(VAR _path: ARRAY OF CHAR): INTEGER;

Clarion: MkDir(*cstring),short,raw,name('_mkdir')
Equivalent to CreateDirectory

rmdir removes the directory specified in the path parameter.

C++: int _rmdir(const char *_path)

Modula-2: _rmdir(VAR _path: ARRAY OF CHAR):INTEGER;

Clarion: RmDir(*cstring),short,raw,name('_rmdir')
Equivalent to RemoveDirectory

chdir Change directory.

C++: int _chdir(const char *_path)

Modula-2: _chdir(VAR _path: ARRAY OF CHAR): INTEGER;

Clarion: ChDir(*cstring),short,raw,name('_chdir')

Clarion Language Utilities

The Clarion Language Utilities refers to a set of prototypes designed to give your existing applications extra functions and flexibility.

These utilities are included as source in the Clarion \LIBSRC folder. To add the prototypes described below to your existing applications, you need only include the CWUTIL.INC file in the Global Map section of your program:

```
INCLUDE('CWUTIL.INC'),ONCE
```

Each utility is described in more detail in its own separate help topic. Use the links below to reference the following utilities:

BeginUnique

BLOBToFile

ByteToHex

CreateDirectory

EndUnique

FileExists

FileToBLOB

FullDrag

GetFileDate

GetFileTime

GetReg

GetTempFileName

GetTempPath

GetUserName

IsTermServer

LongToHex

OSVersion

PutReg

RemoveDirectory

ShortToHex

ValidateOLE

WindowExists

BeginUnique (Set Application to Run in a Single Process)

BeginUnique(*applicationname* **)**

BeginUnique	Sets an application to run as a single process
applicationname	A string constant or variable that specifies name of your application. Example: 'INVOICE.EXE'

BeginUnique returns FALSE if the program specified in *applicationname* is already running (active). If not running, **BeginUnique** returns an event number specified by Windows. This event number can be used by the **EndUnique** statement to terminate the single process mode.

To add this function to your existing applications, you need only include the CWUTIL.INC file in the Global Map section of your program:

```
INCLUDE('CWUTIL.INC'),ONCE
```

Return Data Type: LONG

Example:

```
IF NOT BeginUnique(GLO:ApplicationName)
 MESSAGE(CLIP(GLO:ApplicationName) & ' already running.')
 RETURN
ELSE
 RUN(GLO:ApplicationName)
END
```

See Also:

EndUnique

BLOBTOFILE (Copy Data from BLOB Field to File)

BLOBTOFILE(*bloblabel, filename* **)**

BLOBTOFILE	Copy the contents of a BLOB field to an external file.
bloblabel	The fully qualified label of the BLOB field. (Example: *Customer.BlobImage*)
filename	A string constant or variable that names the output file where the BLOB will be copied.

BLOBTOFILE is used to copy the contents of a BLOB to an external file. If the copy fails for any reason, BLOBTOFILE returns the ERRORCODE posted.

BLOBTOFILE (and FILETOBLOB) are simply binary-to-binary operations.

If you need to save images to a BLOB, and later restore them to an output file, the type of image should also be saved in the database (JPG, GIF, BMP, etc.). Using BLOBTOFILE to save to a different extension can produce unpredictable results.

To add this function to your existing applications, you need only include the CWUTIL.INC file in the Global Map section of your program:

```
INCLUDE('CWUTIL.INC'),ONCE
```

Return Data Type: SIGNED

Example:

```
IF BLOBTOFILE(CUS:ImageBlob, 'imagename.jpg')  !returns an ERRORCODE if copy fails
  MESSAGE('BLOB did not copy due to the following ERRORCODE: ' & ERRORCODE())
END
```

See Also:

FILETOBLOB

BLOB

BYTETOHEX (convert a BYTE to Hexadecimal)

BYTETOHEX(*number, flag* **)**

BYTETOHEX	Convert a BYTE value to its Hexadecimal equivalent.
number	A BYTE variable or constant
flag	A BYTE used to designate a lower or upper case HEX symbol (A,B,C,D,E)

BYTETOHEX is used to convert a *number* to its Hexadecimal equivalent. If the *flag* variable is non-zero, any non-numeric Hexadecimal symbols are returned in lowercase. If zero (default), the non-numeric digits are returned in uppercase.

To add this function to your existing applications, you need only include the CWUTIL.INC file in the Global Map section of your program:

```
INCLUDE('CWUTIL.INC'),ONCE
```

Return Data Type: STRING

Example:

```
BYTETOHEX(255,0)    !returns 'FF'
BYTETOHEX(255,1)    !returns 'ff'
```

See Also:

SHORTTOHEX

LONGTOHEX

CreateDirectory (Create a directory)

CREATEDIRECTORY(*directoryname* **)**

CREATEDIRECTORY	Create a new directory
directoryname	A string constant or variable that stores the directory name

CREATEDIRECTORY creates a new directory with the name passed in the *directoryname* parameter. CREATEDIRECTORY returns zero (0) if successful, and non-zero if not. You can query the ERRNO built-in function to trap for the following error codes:

3 – Path not found (One of the higher path components in *directoryname*)

5 – Access Denied (Possible Security Rights)

183 – Directory Already Exisits

Note:

On some Windows versions, any attempt to create multiple levels of directories
(For example. ‘C:\dir1\dir2\dir3’) will fail, but the error code will not be returned correctly. **CREATEDIRECTORY** will still post a non-zero value, which you can use to trap and post a generic "Directory Not Created" error.

To add this function to your existing applications, you need only include the CWUTIL.INC file in the Global Map section of your program:

```
INCLUDE('CWUTIL.INC'),ONCE
```

Return Data Type: BYTE

Example:

```
PROGRAM


GLO:NewDirectoryName    STRING(200)


 MAP
  INCLUDE('CWUTIL.INC'),ONCE
  MODULE('')
   errno(),*SIGNED,NAME('__errno__')   !prototype built-in error flag
  END
 END


!Make sure project define has _ABCLinkMode_=>1


  CODE
  GLO:NewDirectoryName = CLIP('C:\Temp')


  IF CreateDirectory(GLO:NewDirectoryName)
   CASE Errno()
   OF 3
    MESSAGE('Path Not Found')
   OF 5
    MESSAGE('Access Denied')
   OF 183
    MESSAGE('Directory Already Exists')
   END
  END
```

See Also:

RemoveDirectory

EndUnique (Close an application's event handle)

EndUnique(*eventnumber* **)**

EndUnique	Closes an application’s event number
eventnumber	A numeric constant or variable that uniquely identifies an application event.

EndUnique is used to invalidate the specified application event handle. This is useful where a function using BeginUnique was no longer valid, and you need to override the single event process when subsequent applications are started.

To add support for this utility to your existing applications, you need only include the CWUTIL.INC file in the Global Map section of your program:

```
INCLUDE('CWUTIL.INC'),ONCE
```

Example:

```
EndUnique(GLO:AppEventNumber)
```

See Also:

BeginUnique

FileExists (Confirm file existence)

FILEEXISTS(*filename* **)**

FILEEXISTS	Confirm the existence of a file
filename	A string constant or variable containing the name of the file (and path, if applicable)

FILEEXISTS confirms the existence of a file. If FILEEXISTS returns TRUE (1), the file exists. If FILEEXISTS returns FALSE (0), the file specified in the *filename* parameter does not exist.

To add support for this utility to your existing applications, you need only include the CWUTIL.INC file in the Global Map section of your program:

```
INCLUDE('CWUTIL.INC'),ONCE
```

Return Data Type: BYTE

Example:

```
IF NOT FILEEXISTS(GLO:NewFile)            !If the file does not exist
 DO CreateFile                            !Call the ROUTINE to create it
END

IF NOT FILEEXISTS('C:\INVOICE\Config.dat')  !Does the config file exists
 InitConfig                                 !Call init procedure
END
```

FILETOBLOB (Copy data from a file to a BLOB field)

FILETOBLOB(*filename, bloblabel* **)**

FILETOBLOB	Copy the contents of a file to a BLOB field.
filename	A string constant or variable that names the input file to copy to a BLOB field.
bloblabel	The fully qualified label of the BLOB field. (Example: *Customer.BlobImage*)

FILETOBLOB is used to copy the contents of a file to a BLOB field. If the copy was unsuccessful, FILETOBLOB returns the ERRORCODE posted.

To add support for this utility to your existing applications, you need only include the CWUTIL.INC file in the Global Map section of your program:

```
INCLUDE('CWUTIL.INC'),ONCE
```

Return Data Type: SIGNED

Example:

```
IF FILETOBLOB(GLO:ImageFilename, CUS:ImageBlob)  !returns an ERRORCODE if copy fails
  MESSAGE(CLIP(GLO:ImageFilename) & ' was not copied -  ERRORCODE: ' & ERRORCODE())
END
```

See Also:

BLOBTOFILE

BLOB

FullDrag (Query/Change Window Drag Setting)

FULLDRAG(< *setdragflag* > **)**

FULLDRAG	Query and/or change the full window drag settings
setdragflag	A BYTE variable or constant. TRUE (1) or FALSE (0)

FULLDRAG returns the current window drag setting. If the optional *setdragflag* is set to TRUE (1), full window dragging is enabled. If the optional *setdragflag* is set to FALSE (0), full window dragging is disabled and only the window frame will appear when dragging a window.

To add support for this utility to your existing applications, you need only include the CWUTIL.INC file in the Global Map section of your program:

```
INCLUDE('CWUTIL.INC'),ONCE
```

Return Data Type: LONG

Example:

```
IF NOT FULLDRAG()               !If full window dragging is OFF
  FULLDRAG(1)                   !Enable it
END
```

GetFileDate (Get the file date)

GETFILEDATE(*filename* **)**

GETFILEDATE	Return the date stamp of a file
filename	A string constant or variable containing the name of the file (and path, if applicable)

GETFILEDATE returns the date stamp of the file specified by the *filename* parameter. The date is returned as a LONG that is deformatted and returned in an @D2 picture format. If the file is invalid or does not exist, **GETFILEDATE** returns a zero (0).

To add support for this utility to your existing applications, you need only include the CWUTIL.INC file in the Global Map section of your program:

```
INCLUDE('CWUTIL.INC'),ONCE
```

Return Data Type: LONG

Example:

```
Filedate = GETFILEDATE(LOC:Filename)
```

GetFileTime (Get the file time)

GETFILETIME(*filename* **)**

GETFILETIME	Returns the time stamp of a file
filename	A string constant or variable containing the name of the file (and path, if applicable)

GETFILETIME returns the time stamp of the file specified by the *filename* parameter. The time is returned as a LONG that is deformatted and returned in an @T4 picture format. If the file is invalid or does not exist, **GETFILETIME** returns a zero (0).

To add support for this utility to your existing applications, you need only include the CWUTIL.INC file in the Global Map section of your program:

```
INCLUDE('CWUTIL.INC'),ONCE
```

Return Data Type: LONG

Example:

```
Filetime = GETFILETIME(LOC:Filename)
```

GETREG(get Windows registry entry)

GETREG(*root*, *keyname* [, *valuename*] [, *? *valuetype*]),STRING

GETREG	Gets the value of a specific key and/or value from the system registry.
Root	A LONG integer, variable or expression that contains the root section of the registry from which to obtain the value. Valid values for this are defined in equates.clw and are as follows:
	REG_CLASSES_ROOT
	REG_CURRENT_USER
	REG_LOCAL_MACHINE
	REG_USERS
	REG_PERFORMANCE_DATA
	REG_CURRENT_CONFIG
	REG_DYN_DATA
keyname	A STRING constant, variable or expression that contains the key name of the key whose value is to be queried. This may contain a path separated by backslash '\' characters.
valuename	A STRING constant, variable or expression that contains the name of the value to be queried, if omitted, the value associated directly with the key is returned.
valuetype	An optional integer variable that receives the type of value. It can be used for correct interpreting of the returned *valuename*.
	Supported types are:
	REG_NONE - value is ignored
	REG_SZ - value is a string
	REG_EXPAND_SZ - value is a string that can contain unexpanded environment variables
	REG_MULTI_SZ - value is a string that consists of substrings separated with '<0>' characters; last substring is terminated with two consecutive '<0>'

characters. NOTE: this value result can not be assigned to a CSTRING

REG_DWORD - value is a 32 bit number

REG_DWORD_LITTLE_ENDIAN - value is a 32 bit number

REG_DWORD_BIG_ENDIAN - value is a 32 bit number

REG_QWORD - value is a 64 bit number

REG_QWORD_LITTLE_ENDIAN - value is a 64 bit number

REG_BINARY - value string is a binary data of any form

The REG_* constants are declared in EQUATES.CLW

The **GETREG** function returns the value of named entry in the system registry as a Clarion string. If the requested entry does not exist, an empty string is returned.

Vista and Clarion Built-in Registry Functions

On Vista, a (non-elevated) app can READ the registry key HKLM (local machine) with no problems, but as far as writing to the Registry it needs to use HKCU (current user).

If you code sign your executable and run with elevated priveleges then you can write to HKLM

Return Data Type: STRING

Example:

```
  PROGRAM

  MAP
  END

  INCLUDE('EQUATES')
CurrentPath CSTRING(100)
ColorScheme CSTRING(100)
   CODE
   CurrentPath =|
   GETREG(REG_LOCAL_MACHINE,'SOFTWARE\SoftVelocity\Clarion7','root')
   !Returns root directory of Clarion 7 install
    ColorScheme =|
   GETREG(REG_CURRENT_USER,'Control Panel\Current','Color Schemes')
   !get the current user's color scheme
```

See Also:

PUTREG, DELETEREG

GetTempFileName (Generate a temporary file)

GETTEMPFILENAME(*prefix, <pathname>* **)**

GETTEMPFILENAME	Returns the name of a temporary file
prefix	A string constant or variable naming the prefix (first three letters) of the temporary file. If blank, the default prefix used is '$$$'
pathname	A string constant or variable naming the location of the temporary file. If omitted, the system TEMP or TMP directory path is used.

GETTEMPFILENAME is used to generate a temporary file. If the *pathname* specified is invalid, **GETTEMPFILENAME** returns an empty string.

Note:

Make sure to remove your temporary files that you create after use. The Windows system will not automatically remove these files.

To add support for this utility to your existing applications, you need only include the CWUTIL.INC file in the Global Map section of your program:

```
INCLUDE('CWUTIL.INC'),ONCE
```

Return Data Type: STRING

Example:

```
!Note ## represents a random number assigned to the temporary file name
    message(GETTEMPFILENAME('bob','d:\help'))    !created 'bob##.tmp' in D:\help
    message(GETTEMPFILENAME(''))                 !created '$$$##.tmp' in
                                                 !C:\WINNT\TEMP (my TEMP path)
```

GetTempPath (Return TMP or TEMP environment path)

GETTEMPPATH( )

GETTEMPPATH Returns the name of the path specified by the Windows Environment variables

GETTEMPPATH is used to return the full path designated by the TMP or TEMP Windows Environment settings.**GETTEMPPATH** returns the first Environment setting it finds.

To add support for this utility to your existing applications, you need only include the CWUTIL.INC file in the Global Map section of your program:

```
INCLUDE('CWUTIL.INC'),ONCE
```

Return Data Type: STRING

Example:

```
   GLO:TempPath = GETTEMPPATH( )   !return environment path
```

GetUserName (Return Network User Name)

GETUSERNAME( )

GETUSERNAME Returns the current default user name

GETUSERNAME is used to retrieve the current default user name, or the user name used to establish a network connection. **GETUSERNAME** returns a blank string if an error is encountered.

To add support for this utility to your existing applications, you need only include the CWUTIL.INC file in the Global Map section of your program:

```
INCLUDE('CWUTIL.INC'),ONCE
```

Return Data Type: STRING

Example:

```
  GLO:LoginName = GETUSERNAME()   !return a unique login name
```

IsTermServer (Verify Terminal Server Usage)

ISTERMSERVER( )

ISTERMSERVER Detects Terminal Server usage

It is a good practice for applications to detect whether they are running in a Terminal Services Client session in order to optimize performance. For example, when an application is running on a remote session, it should eliminate unnecessary graphic effects. If a user is running the application directly on the terminal, it is not necessary for the application to optimize its behavior.

ISTERMSERVER is used to detect Terminal Server usage by returning the status of the System Metrics SM_REMOTESESSION flag. **ISTERMSERVER** returns TRUE if an application is running in a Terminal Services Client session, and FALSE if the application is running on the console.

This function is only valid for Windows 2000 or later.

To add support for this utility to your existing applications, you need only include the CWUTIL.INC file in the Global Map section of your program:

```
INCLUDE('CWUTIL.INC'),ONCE
```

Return Data Type: BYTE

Example:

```
GLO:RemoteSessionActive = ISTERMSERVER() !is a remote session active?
```

LONGTOHEX (convert an unsigned LONG to Hexadecimal)

LONGTOHEX(*number, flag* **)**

LONGTOHEX	Convert a ULONG value to its Hexadecimal equivalent.
number	A ULONG variable or constant
flag	A BYTE used to designate a lower or upper case HEX symbol (A,B,C,D,E)

LONGTOHEX is used to convert a *number* to its Hexadecimal equivalent. If the *flag* variable is non-zero, any non-numeric Hexadecimal symbols are returned in lowercase. If zero (default), the non-numeric digits are returned in uppercase.

To add support for this utility to your existing applications, you need only include the CWUTIL.INC file in the Global Map section of your program:

```
INCLUDE('CWUTIL.INC'),ONCE
```

Return Data Type: STRING

Example:

```
LONGTOHEX(32000000,0)    !returns 1E84800
LONGTOHEX(32000000,1)    !returns 1e84800
```

See Also:

BYTETOHEX

SHORTTOHEX

PROP:WindowsVersion

Returns the string that describes Windows version running the program.

The runtime library returns the following information:

Windows/Win32 <version> [<category> | <edition>] [ServicePack] Build Number

Currently, values up to the *ServicePack* can be one of the following strings:

Windows 95
Windows 95 OSR2
Windows 98
Windows 98 SE
Windows Me
Windows NT 3.5 WorkStation
Windows NT 4 WorkStation
Windows 2000 Professional
Windows XP Home Edition
Windows XP Professional
Windows XP Tablet PC Edition
Windows XP Media Center Edition
Windows NT 3.5 Server
Windows NT 3.5 Advanced Server
Windows NT 4 Server
Windows NT 4 Server, Enterprise Edition
Windows 2000 Advanced Server
Windows 2000 Small Business Server
Windows 2000 Datacenter Server
Windows Server 2003 Standard Edition
Windows Server 2003 Datacenter Edition
Windows Server 2003 Enterprise Edition
Windows Server 2003 Web Edition
Windows Vista Professional Build xxxx

The Service Pack value is present only if applicable

Read only. Available for SYSTEM only.

Example:

```
GLO:WindowsVersion = SYSTEM{PROP:WindowsVersion}
```

PUTREG (write value to Windows registry)

PUTREG(*root*, *keyname*, *valuename* [,*value*] [,*valuetype*] **)**

PUTREG	Writes a string value into the system registry.
root	A LONG integer constant, variable or expression that specifies the root section of the registry to which to write the value. Valid values for this are defined in **equates.clw** and are as follows: **REG_CLASSES_ROOT** **REG_CURRENT_USER** **REG_LOCAL_MACHINE** **REG_USERS** **REG_PERFORMANCE_DATA** **REG_CURRENT_CONFIG** **REG_DYN_DATA**
keyname	A STRING constant, variable or expression that contains the key name of the key whose value is to be written. This may contain a path separated by backslash '\' characters. The *keyname* cannot contain a leading backslash.
valuename	A STRING constant, variable or expression that contains the name of the value to be written.
value	A STRING constant, variable or expression that contains the value to be written to the registry in the position given. If omitted, an empty string is written to the registry.
valuetype	A LONG integer constant, variable or expression that specifies the how to store passed value. If omitted, the default value is REG_SZ (see below). Supported types (expressed as EQUATES):

REG_NONE	value is ignored
REG_SZ	value is stored as a null terminated UNICODE string (default)
REG_EXPAND_SZ	value is stored as a null terminated UNICODE string that can contain unexpanded environment variables.

REG_MULTI_SZ	value is stored as an array of null terminated UNICODE strings. Strings in the array (i.e. in value parameter) are separated with '<0>' characters. GETREG appends 2 '<0>' characters at the end of array if they are not given.
REG_DWORD	value is stored as a 32 bit number, lower bytes stored first, e.g. value equal to 12345678h is stored as '<78h,56h,34h,12h>'
REG_DWORD_LITTLE_ENDIAN	the same as REG_DWORD
REG_DWORD_BIG_ENDIAN	value is stored as a 32 bit number with backward order of bytes, e.g. value equal to 12345678h is stored as '<12h,34h,56h,78h>'
REG_QWORD	value is a 64 bit number, lower bytes are stored first
REG_QWORD_LITTLE_ENDIAN	the same as REG_QWORD
REG_BINARY	value string is a binary data of any form

The PUTREG procedure places the value into a valuename that exists in the Windows registry. The key and value will be created if it does not exist. If the write to the registry is successful, PUTREG returns 0. If the write to the registry is unsuccessful, the result is a non-zero value.

Vista and Clarion Built-in Registry Functions

On Vista, a (non-elevated) app can READ the registry key HKLM (local machine) with no problems, but as far as writing to the Registry it needs to use HKCU (current user).

If you code sign your executable and run with elevated priveleges then you can write to HKLM

Return Data Type: LONG

Example:

```
  PROGRAM

   MAP.

  INCLUDE('EQUATES')
CurrentPath CSTRING(100)
ColorScheme CSTRING(100)

    CODE
    !Set the root directory of Clarion 7 install
    CurrentPath = 'C:\Clarion7'
    IF PUTREG(REG_LOCAL_MACHINE,'SOFTWARE\SoftVelocity\Clarion7','root',CurrentPath)
     MESSAGE('Unable to set the root directory') !post error if non-zero value returned
    END
    !writes the current user's color scheme to the registry
    ColorScheme = 'Windows Standard'
    IF PUTREG(REG_CURRENT_USER,'Control Panel\Current','Color Schemes',ColorScheme)
     MESSAGE('Unable to set the color scheme')   !post error if non-zero value returned
    END
```

See Also:

GETREG, DELETEREG

RemoveDirectory (Remove a directory)

REMOVEDIRECTORY(*directoryname* **)**

REMOVEDIRECTORY	Remove an existing directory
directoryname	A string constant or variable that stores the directory name

REMOVEDIRECTORY removes an existing directory with the name passed in the *directoryname* parameter. **REMOVEDIRECTORY** returns zero (0) if successful, and non-zero if not. You can query the ERRNO built-in library function to trap for the following error codes:

3 – Path not found (One of the higher path components in *directoryname*)

5 – Access Denied (Path may refer to a file, root directory, or current directory)

To add support for this utility to your existing applications, you need only include the CWUTIL.INC file in the Global Map section of your program:

```
INCLUDE('CWUTIL.INC'),ONCE
```

Return Data Type: BYTE

Example:

```
  MODULE('')
    errno(),*SIGNED,NAME('__errno__')   !proptotype built-in error flag
  END

IF REMOVEDIRECTORY(GLO:NewDirectoryName)
 CASE Errno()
 OF 3
  MESSAGE('Path Not Found')
 OF 5
  MESSAGE('Access Denied')
 END
END
```

See Also:

CreateDirectory

ResizeImage (Resize an image to fit a control)

RESIZEIMAGE(*Control, XPos, YPos, Width, Height, <Report>)*

RESIZEIMAGE	Resize a valid graphic file to fit inside a target IMAGE control
Control	The Field Equate Label of the target IMAGE control.
Xpos	A SHORT constant or variable identifying the horizontal position of the top left corner of the target IMAGE control in dialog units.
Ypos	A SHORT constant or variable identifying the vertical position of the top left corner of the target IMAGE control in dialog units.
Width	A SHORT constant or variable identifying the width of the target IMAGE control in dialog units.
Height	A SHORT constant or variable identifying the height of the target IMAGE control in dialog units.
Report	A valid label of a REPORT structure. Indicates that the control to store the resized image is contained in a REPORT target instead of a WINDOW

RESIZEIMAGE is used to resize the image to fit the original control size. If an image is larger than the target control, the image will be reduced to fit the target control's position parameters. If an image is smaller than the target control, the image will be expanded to fit the target control's position parameters.

To add support for this utility to your existing applications, you need only include the CWUTIL.INC file in the Global Map section of your program:

```
INCLUDE('CWUTIL.INC'),ONCE
```

Example:

```
CASE ACCEPTED()
 OF ?LookupFile
   ThisWindow.Update
   LOC:Filename = FileLookup9.Ask(0)
   DISPLAY
   IF LOC:Filename
      ?Image1{PROP:TEXT} = LOC:Filename      !Move filename to image field
      ResizeImage(?Image1,114,132,90,64)     !Resize it
   END
 OF ?OK
   ThisWindow.Update
   IF SELF.Request = ViewRecord AND NOT SELF.BatchProcessing THEN
      POST(EVENT:CloseWindow)
   END
END
```

SHORTTOHEX (convert an unsigned SHORT to Hexadecimal)

SHORTTOHEX(*number, flag* **)**

SHORTTOHEX	Convert a USHORT value to its Hexadecimal equivalent.
number	A USHORT variable or constant
flag	A BYTE used to designate a lower or upper case HEX symbol (A,B,C,D,E)

SHORTTOHEX is used to convert a *number* to its Hexadecimal equivalent. If the *flag* variable is non-zero, any non-numeric Hexadecimal symbols are returned in lowercase. If zero (default), the non-numeric digits are returned in uppercase.

To add support for this utility to your existing applications, you need only include the CWUTIL.INC file in the Global Map section of your program:

```
INCLUDE('CWUTIL.INC'),ONCE
```

Return Data Type: STRING

Example:

```
SHORTTOHEX(64000,0)  !returns 'FA00'
SHORTTOHEX(64000,1)  !returns 'fa00'
```

See Also:

BYTETOHEX LONGTOHEX

ValidateOLE (Validate OLE Control Creation)

VALIDATEOLE(*OLEControl, < OLEFileName> , <OLECreateName>* **)**

VALIDATEOLE	Validate that an OLE control has been successfully created
OLEControl	A field number or field equate label of the OLE control.
OLEFileName	(under construction)
OLECreatename	(under construction)

VALIDATEOLE is used to verify that an OLE control has been created successfully. **VALIDATEOLE** returns TRUE if the OLE control has been successfully created.

If not successful, **VALIDATEOLE** can optionally display a message box that describes why the OLE control could not be created, provided that the *OLEFilename* parameter is passed, and then returns FALSE. Otherwise, VALIDATEOLE just returns FALSE if only the *OLEControl* is designated.

To add support for this utility to your existing applications, you need only include the CWUTIL.INC file in the Global Map section of your program:

```
INCLUDE('CWUTIL.INC'),ONCE
```

Return Data Type: BYTE

Example:

```
LOC:OLEActive = VALIDATEOLE( ) !is a remote session active?
```

WindowExists (Validate Window Existence)

WINDOWEXISTS(*windowtitle* **)**

WINDOWEXISTS	Verify that a WINDOW structure is active
windowtitle	A string constant or variable that specifies the window name (the window's title).

WINDOWEXISTS is used to verify that a valid window exists whose window name matches the window title.

If **WINDOWEXISTS** succeeds (has a valid handle), the return value is TRUE (1).

If it fails, the return value is FALSE (0).

To add support for this utility to your existing applications, you need only include the CWUTIL.INC file in the Global Map section of your program:

```
INCLUDE('CWUTIL.INC'),ONCE
```

Return Data Type: BYTE

Example:

```
GLO:IsMenuActive = WINDOWEXISTS('Utility Menu') !is the utility window active?
```

Clarion 7 Build System

Clarion 7 Build System

The Clarion build system in Version 7 supports both the legacy (Clarion 6 and prior versions) project system, plus MSBuild (Microsoft) project files. MSBuild project files are supported via two custom MSBuild Tasks: CW and CWClean. The CWClean task is used to remove files created by the CW task. The CW task creates a temporary legacy project file and passes that to the Clarion Project System.

Clarion Project System Version 7

The Clarion Project System Version 7 is a complete rewrite of the TopSpeed Project System used in previous versions of Clarion. It supports almost all commands that the TopSpeed Project System supported, plus adds new features to make it easier to build large systems.

Automatically Included Projects

When a project file is included the project system will first look for a file called *solglobals.pi*. If this exists it is included at the top of the project. This file is intended to be used to contain solution wide project settings. For example. The file may contain the line:

```
#pragma define (_MyProductVersion_=>10)
```

After *solglobals.pi* has been included, the project system will then add *cwglobals.pi*. This file is intended to be used to contain company wide project settings. For example:

```
#pragma define(_Company_=>Our Company Name)
```

Project Commands

#section

```
#section <name>
-- section commands
#else <name>
-- section commands
#else <name>
-- section commands
#endsection
```

A section starts a list of commands that are activated on a **#activate** command. You can can have one or mutually exclusive sections by specifying a **#else** command. If a **#activate** command is encountered and a mutually exclusive section had previous been activated via a **#activate** command, then the previous command is deactivated without the need to call **#deactivate**.

Only #set and #pragma commands are valid within a section.

Example:

```
-- Contents of cwglobal.pi
#section release
#set outdir=release
#else debug
#pragma debug(line_num=>on)
#pragma check(stack=>on)
#pragma check(index=>on)
#set outdir=debug
#endsection

#section Profile
#set profiling = on
#endsection
#set profiling = off

#section Tracing
#set tracing = on
#pragma define(_TRACING_=>on)
#endsection
#set tracing = off

#activate release -- same as doing #set outdir=release
#activate Profile -- same as doing #set profile = on
#activate debug -- #deactivate release done automatically
```

#activate

#activate <*sectionname*>

Activates a set of commands previously defined by the #section command.

If the section being activated is part of a list of mutually exclusive sections, and one of these sections is currently active, then an implicit **#deactivate othersection** is called.

When an **#activate** command is encountered then all **#set** and **#pragma** statements contained within that section are executed.

#deactivate

#deactivate <*sectionname*>

Reverts all commands that where activated by a previous call to **#activate sectionname**.

This command will set the pragmas and project sets back to the value they where at before the matching #activate call was made.

#active (Boolean Operator)

#active SectionName

Returns TRUE (1) if the SectionName is currently active.

Obsolete Commands:

The following project commands are no longer supported:

Command	Remarks
#exemod	This command was for 16 bit only projects
#getkey	There is no way to pause a build
#ignore	This optimization is no longer supported
#prompt	Use #message
#rundll	Integrated compilers are known to the project system now
#scan	This optimization is no longer required

MSBuild tasks

.cwproj File Format

The .cwproj file is a standard MSBuild file in a format that allows the IDE to provide a graphical interface on top of the file. This file consists of properties and items that are passed to the CW and CWClean tasks.

The following shows the properties and items in the .cwproj file, and which task parameter they map to.

Item/Property	Item or Property	CW Task Parameter	CWClean Task Parameter
Compile	Item	Sources	
FileDriver	Item	FileDrivers	
Include	Item	Includes	
Library	Item	Libraries	
None	Item	CopyableFiles	
ProjectReference	Item	References	
Reference	Item	References	
ApplicationIcon	Property	Icon	
check_index	Property	CheckIndex	
check_stack	Property	CheckStack	
clarion_version	Property	Version	Version
CopyCore	Property	CopyCoreFiles	
cpu	Property	CPU	
DefineConstants	Property	Defines	
GenerateMap	Property	MapFile	
line_numbers	Property	LineNumbers	
Model	Property	Model	
OutputName	Property	TargetName	
OutputPath	Property	TargetDir	
OutputType	Property	TargetType	

RedFile	Property	RedirectionFile
stack_size	Property	StackSize
vid	Property	Vid

Which task is invoked is dependent on the build target. If a Build is being performed (the default target), then the CW Task is invoked. If a Clean is being performed, then the CWClean Task is invoked. If a Rebuild is being performed, then both the CWClean Task and CW Task are invoked.

CW Task

Invokes the Clarion project system which produces executables (.exe), dynamic-link libraries (.dll), or code libraries (.lib).

Parameters

The following table describes the parameters of the **CW** task.

Parameters	Description
CheckIndex	Optional **Boolean** parameter. If true, then compiler generates check code when array indexes are used Equivalent to #pragma check(index=>value) in the legacy project system
CheckStack	Optional **Boolean** parameter. If true, then compiler generates stack access checking code Equivalent to #pragma check(index=>value) in the legacy project system
CopyableFiles	Optional **ITaskItem[]** parameter. Specifies one or more files that should be copied to the destination directory. This list is only processed if **TargetType** is Exe
CopyCoreFiles	Optional **Boolean** parameter. If true, the core DLLs will be copied
CPU	Optional **String** parameter. Indicates which CPU the code is optimized for. Currently ignored.

Defines	Optional **String** parameter. Specifies a list of defines set when building this project. This is a semi-colon ';' separated list of items in the format item=>value. As the .cwproj file is an MSBuild file, this list must use standard XML notation. So a list of 2 elements _ABCDllMode_=>on;_ABCLinkMode_=>off will look like <defines>_ABCDllMode_=&gt;on%3b_ABCLinkMode_=&gt;off </defines> This is equivalent to #pragma define(define=>value;define=>value) in the legacy project system
FileDrivers	Optional **ITaskItem[]** parameter. Specifies one or more File Drivers that should be linked into the project.
Icon	Optional **String** parameter. Indicates the name of the file that contains the icon that Windows will associate with the Exe. Equivalent to #pragma link_option(icon => iconname) in the legacy project system
Includes	Optional **String** parameter. Semi-colon ';' separated list of legacy project files to be included into this project. Equivalent to #include(projectFile) in the legacy project system
LibFile	Optional **String** output parameter. Specifies the .lib file that was created by this project
Libraries	Optional **ITaskItem[]** parameter. Specifies one or more .lib files that should be linked into the project. Equivalent to #pragma link (libfile) in the legacy project system
LineNumbers	Optional **Boolean** parameter. If true, then line number information will be added to the MAP file Equivalent to #pragma debug (line_num=>value) in the legacy project system
MapFile	Optional **Boolean** parameter. If true, then line number information will be added to the MAP file Equivalent to #pragma link_option (map=>value) in the legacy project system
Model	Optional **String** parameter. Specifies the memory model. Valid values are Dll, Lib, and CustomDll
OutputFile	Optional **String** output parameter. Specifies the .exe or .dll file that was created by this project

ProjectName	Required **String** parameter.
	Specifies the name of the project file being built. Normally set to the built-in project property **MSBuildProjectFile**
RedirectionFile	Optional **String** parameter.
	Specifies the redirection file to be used for this build.
	If not supplied, then the default redirection file is used based on the directory of the project and the **Version** parameter
References	Optional **ITaskItem[]** parameter.
	Specifies one or more legacy or MSBuild project files that should be used to obtain .lib files that this project uses.
	Any project in this list will be built when you build this project. Any .lib file created by the referenced project will be added to the **Libraries** list.
RequiredDlls	Optional **ITaskItem[]** output parameter.
	Specifies the list of dlls that this project depends on.
Sections	Optional **String** parameter.
	Specifies which sections are to be active on start of the project.
	Normally set to the built in property **Configuration**.
	Controls which parts of the redirection file are active during the build.
	Can also be used to activate sections defined in the cwglobals.pi or solglobals.pi files.
Sources	Required **ITaskItem[]** parameter.
	Specifies the list of source files to be compiled in this project
	Equivalent to #compile (file) in the legacy project system
	Alternatively can contain 1 .pr or .prj file. In this case that legacy project file is built.
StackSize	Optional **Int** parameter.
	Specifies the size of the stack.
	Equivalent to #pragma link_option (stack=>value) in the legacy project system
TargetDir	Optional **String** parameter.
	Specifies the directory where the target file will be created.
	If specified this overrides any settings in the redirection file.
TargetName	Optional **String** parameter.
	Specifies the name of the exe, dll or lib being created by the project.
	This must be specified unless there is only 1 item in the sources list and this is a .prj or .pr file.

TargetType	Optional **String** parameter. Indicates what type of binary is being created. Valid values are: ‘dll’ - indicates that a .DLL file is being created ‘lib’ - indicates that a .LIB file is being created Anything else will be treated as ‘exe’ and a .EXE will be created.
Version	Optional **String** parameter. Indicates which version of Clarion to use to build the project. Default is to use the current version of Clarion
Vid	Optional **String** parameter. Indicates what debug support should be compiled into the .obj files Valid values are full, min, off.

CopyableFiles Meta Data

Any item in the **CopyableFiles** list can optionally have the following Metadata elements

Element	**Description**
CopyToOutputDirectory	Optional **String** element. Indicates if the file should be copied to the destination directory of the Exe file. If set to "always" then the file will always be copied. If set to "preservenewest" then the file will be copied only if it is newer than the file in the destination directory

Libraries Meta Data

Any item in the **Libraries** list can optionally have the following Metadata elements

Element	**Description**
CopyToOutputDirectory	Optional **String** element. Indicates if the file should be copied to the destination directory of the Exe file. If set to "always" then the file will always be copied. If set to "preservenewest" then the file will be copied only if it is newer than the file in the destination directory
DllLocation	Optional **String** element. Only tested if **CopyToOutputDirectory** is not set to "always" or

	"preservenewest". Indicates where the dll that matches this library can be found. If not specified, then the dll is assumed to be in the format <library_name>.dll and can be found via redirection.
CopyDllToOutputDirectory	Optional **String** element. Only tested if **CopyToOutputDirectory** is not set to "always" or "preservenewest". Indicates when the dll that matches this library should be copied. If set to "always" then the dll will always be copied. If set to "preservenewest" then the dll will be copied only if it is newer than the dll in the desination directory. Otherwise it is copied if the CW Task parameter **CopyCoreFiles** is true.

References Meta Data

Any item in the **References** list can optionally have the following Metadata elements

Element	Description
CopyReferencesToOutputDirectory	Optional **String** element. Valid values are "Always", "Never", or "Default". Default value is "Default". Only applies to referenced .cwproj files. Indicates if referenced dlls should be copied to the output directory. A value of Default indicates that referenced dlls should be copied if **CopyCoreFiles** is true.
CopyToOutputDirectory	Optional **Boolean** element. Default value is false. Only applies to referenced .cwproj files. If false, then files created by this project or flagged to be copied will not be added to the list of files to be copied.
Link	Optional **Boolean** element. Default value is true. If the reference is a .cwproj file and **link** is false, then files created by this project or flagged to be copied will not be added to the list of files to be copied. If the reference is a legacy project file, then it is not included in the current build.

Sources Meta Data

Any item in the **Sources** list can optionally have the following Metadata elements

Element	Description
check_index	Optional **Boolean** element. Indicates if index checking should be generated for this source file. Equivalent to /check(index=>value) in the legacy project system
check_stack	Optional **Boolean** element. Indicates if stack checking should be generated for this source file. Equivalent to /check(stack=>value) in the legacy project system
CopyToOutputDirectory	Optional **String** element. Indicates if the source file should be copied to the destination directory of the Exe file. If set to "always" then the source file will always be copied. If set to "preservenewest" then the source file will be copied only if it is newer than the file in the destination directory
Defines	Optional **String** element. Specifies a list of defines set when compiling this source file. This is a semi-colon ';' separated list of items in the format item=>value. As the .cwproj file is an MSBuild file, this list must use standard XML notation. So a list of 2 elements a=>2;b=>3 will look like: <defines&gt;a=&gt;2%3bb=&gt;3 </defines&gt; This is equivalent to /define(define=>value;define=>value) in the legacy project system
Line_num	Optional **Boolean** element. Indicates if line numbers should be generated in the map for this source file. Equivalent to /debug(line_num=>value) in the legacy project system
To	Optional **String** element. Indicates the name of the object file the source file should be compiled to. Equivalent to the #to option in the legacy project system

Vid	Optional **String** element. Valid values are "full", "min", "off". Indicates what level of debug code to generate for this source file. Equivalent to /debug(vid=>value) in the legacy project system

CWClean Task

Deletes all files created by a previous call to the **CW** task. This task relies on the <projectnam>.FileList.xml file that is created by the CW task.

Parameters

The following table describes the parameters of the **CWClean** task.

Parameters	**Description**
ProjectName	Required **String** parameter. Specifies the name of the project file being built. Normally set to the built-in project property **MSBuildProjectFile**
Sections	Optional **String** parameter. Normally set to the built in property **Configuration**. Controls which parts of the redirection file are active during the clean up.
Version	Optional **String** parameter. Indicates which version of Clarion to use during the clean up. Default is to use the current version of Clarion

Controlling the IDE from the Command Line

User's Guide to ClarionCL.exe (command line interface to the Clarion IDE)

The **ClarionCL** executable allows you to execute some IDE tasks from the command line without the need to start the Clarion IDE.

The syntax for ClarionCL usage is as follows:

ClarionCL [*switch* [*parameter* [*parameter* […]]][*switch* [*parameter* [*parameter* […]]] […]]

Switch Formats

Valid *switch* formats are as follows":

```
/id
Example: C:\Clarion7\BIN\clarionCL /?
-id
Example: C:\Clarion7\BIN\clarionCL -?
--id
Example: C:\Clarion7\BIN\clarionCL --?
```

Valid Switches:

Id	Parameters	Description
Fr	Driver DLL	Registers a file driver
Tl		Displays a list of all registered templates

Tr	Template name	Attempts to register the template file passed as the parameter. The redirection file will be used to find the template file if the file name is supplied without a full path.
		For example, to register the ADO templates for ABC provide the name of the template file:
		`ClarionCL -tr \C70\template\win\abado.tpl`
		The -tr switch uses the redirection (*.red) file, so you do not have to supply the full path if the template file can be found using the redirection system:
		`ClarionCL -tr abado.tpl`

Tu	Template name	Attempts to unregister the template class passed as the parameter. For example, to unregister the ADO templates from both ABC and Clarion chains provide the template class names:

		`ClarionCL -tu ABC_ADO`
		`ClarionCL -tu Clarion_ADO`

?		Display a list of all switches supported by ClarionCL

Note:

The *id* is case insensitive!

Usage Notes and Examples

1. Running ClarionCL without any parameters is equivalent to:

```
ClarionCL /?
```

2. Passing an invalid switch will cause ClarionCL to behave as though `ClarionCL /?` was called.

Examples:

```
ClarionCL -tr mytemplate.tpl
```

Attempts to register *mytemplate.tpl*

```
ClarionCL -tu mytemplate.tpl
```

Attempts to unregister *mytemplate.tpl*

Exit Code

ClarionCL will return an exit code of 0 if no errors occurred. Otherwise the exit code is set to the total number of errors that occurred during the run.

Error Codes

Error Code	Description
CLCE001	An Unhandled Exception was thrown
CLCE002	There were no switches registered via any .addin file
CLCE003	Invalid parameters for the method
CLCE004	Not enough parameters where supplied to run the command. The switch is ignored
CLCE005	Unknown switch

Warning Codes

Error Code	Description
CLCW001	A duplicate switch was passed, but the switch can only appear once. Only the first occurrence of the switch is processed
CLCW002	Too many parameters were supplied for the command. All extra parameters are ignored
CLCW003	The first parameter to ClarionCL does not begin with / or -

Legacy (Pre-Clarion 7) Project System Reference

Introduction

The Project System is integrated into the Clarion Environment. It is a powerful sequential language that combines the functionality of a batch processor, a linker and an intelligent compile-and-link system.

The Project System gives you total control over the compile and link process for the simplest single .EXE project up to the most complicated multiple .DLL project.

The primary benefits of using the Project System are automation, efficiency, and accuracy. With a single command, you can remake your entire project, no matter how complicated, and you can be assured that the correct source and objects are included in the compile and link processes, plus, the components that don't need it, don't get reprocessed. In addition, you can make different versions of your project (release version, debug version, evaluation/demo version, etc.) with the flip of a switch.

Here is a simple example of some project system language generated by the Clarion Application Generator:

```
#noedit
#system win
#model clarion dll
#pragma debug(vid=>full)
#compile QWKTU_RD.CLW— GENERATED
#compile QWKTU_RU.CLW— GENERATED
#compile QWKTU_SF.CLW— GENERATED
#compile QWKTUTOR.clw /define(GENERATED=>on)— GENERATED
#compile QWKTU001.clw /define(GENERATED=>on)— GENERATED
#compile QWKTU002.clw /define(GENERATED=>on)— GENERATED
#compile QWKTU003.clw /define(GENERATED=>on)— GENERATED
#pragma link(C%L%TPS%S%.LIB)— GENERATED
#link QWKTUTOR.EXE
```

Language Components

Keywords start with a pound sign (#). In the example, each keyword begins on a new line for readability. This is not required.

Comments start with a double hyphen (--) and are terminated by a Carriage Return or Line Feed.

Macros are surrounded by percent signs (%). You may want to think of macros as variables—a value is substituted whenever the project system encounters a macro name surrounded by percent signs (%). See Project System Macros.

Keyword Parameters are everything else you see in the example. Parameters and their syntax are discussed with each keyword.

The Project System recognizes the following keywords:

```
#abort #expand #older
#and #file #or
#autocompile #if #pragma
#compile #ignore #prompt
#declare_compiler #implib #run
#dolink #include #rundll
#else #link #set
#elsif #message #split
#endif #model #system
#error #noedit #then
#exemod #not #to
#exists
```

Files and Editing

With regard to Clarion, the project system commands are generally stored in either a .PRJ file or an .APP file. APP files are maintained strictly through Clarion's development environment, however, PRJ files are simple ASCII text and may be maintained with the development environment (See the User's Guide; Using the Project System) or with your favorite text editor.

#noedit

The #noedit command can be placed at the top of a project file to prevent menu-editing from the SoftVelocity environment. It has no effect in the Clarion environment.

Project System Macros

Macros are special strings that indicate a variable substitution is required. You may find it useful to think of macros as variables.

A sequence of characters enclosed by % characters indicates a macro name. The following characters are permitted in macro names:

A B C D E F G H I J K L M

N O P Q R S T U V W X Y Z

a b c d e f g h i j k l m

n o p q r s t u v w x y z

0 1 2 3 4 5 6 7 8 9 _

The trailing % may be omitted provided the character following the macro name is not one of the characters above.

Whenever a % delimited macro name is encountered, it is replaced either by the string associated with that macro, or by an empty string if there is no associated string. Substitution strings are associated with a macro by using the #set command.

Two adjacent % characters may be used when a % character is required in the substituted string. This double % technique can be used to delay macro substitution. For example:

```
#set echo = '#message %%mymac' --'#message %mymac' associated with echo

#set mymac = 'Hello' --'Hello' associated with mymac

%echo --'#message %mymac' substituted for %echo
-- and 'Hello' substituted for %mymac

#set mymac = 'World' --'World' associated with mymac

%echo --'#message %mymac' substituted for %echo
-- and 'World' substituted for %mymac
```

If a single % had been specified in the first #set command, the macro %mymac would have been expanded (to the empty string) before defining the replacement text for the macro %echo. The double % results in the project system executing:

```
#message Hello
#message World
```

The single % results in the project system executing:

```
#message ""
#message ""
```

Setting Macro Values

#set

#set macroname = string

The #set command associates a macro name with a string. Any previous setting for the given macro is lost. The macro name in the #set command should not be delimited by % characters. The string should be enclosed in single quotes if it contains embedded spaces or project system keywords.

For example:

```
#set cwindow = TopSpeed
#set linkit = '#link myfile'
...
#if '%cwindow'= TopSpeed #then
#pragma link(CS_GRAPH.LIB)
#endif

%linkit
```

#expand <file-name>

The filename is subjected to redirection analysis, and the following macros are defined:

%cpath Is set to the fully expanded filename where the file would be created.

%opath Is set to the fully expanded filename where the file would be opened.

%ext Is set to the extension of the filename.

%tail Is set to the filename, less extension, drive and path.

%cdir Is set to the directory where the file would be created.

%odir Is set to the directory where the file would be opened for read (if the file does not exist %opath is set the same as %cpath).

For example, suppose the redirection file has the line,

```
*.def : . ; c:\ts\include
```

and the file c:\ts\include\io.def exists, and the current directory is d:\test then,

```
#expand io.def
```

is equivalent to,

```
#set opath = d:\test\io.def
#set cpath = c:\ts\include\io.def
```

```
#set ext = .def
#set tail = io
#set odir = d:\test\
#set cdir = c:\ts\include\
```

#split <filename>

The filename is split into its base and extension. The following macros are defined:

%ext Is set to the extension of the filename.

%name1 Is set to the filename, less extension.

For example:

```
#split d:\name.exe
```

is equivalent to,

```
#set ext = exe
#set name = d:\name
```

Special Project System Macros

A number of macros are used for special purposes by the Project System, and you should avoid defining macros of the same name inadvertently. Similarly, you should not define macros using trailing underbars.

The following is a list, in alphabetical order, of all such macros:

%action	Set to make, link, compile or run, depending on the mode of invocation.
%cdir	Set by the **#expand** command.
%compile_src	In compile mode, this is set to the name of the file to be compiled, with path and extension where available. Otherwise, it is set to the empty string.
%cpath	Set by the **#expand** command.
%devsys	Set by the Clarion environment to win.
%editfile	Set to the name of the file being edited in the topmost window. If no window is open, or in batch mode, it is set to the empty string.
%editwin	Set to the window number (0-9) of the topmost window. If no window is open, or in batch mode, it is set to the empty string.
%errors	Count of errors produced by preceding compile or **#file adderrors** command.
%ext	Set by the **#split** and **#expand** commands.
%filetype	Set by the **#system** command to its second argument, and examined by the **#link** command.
%jpicall	Set by the **#model** command to its second argument, and examined by the **#link** command.
%L	Set by the **#model** command to either '' (standalone) L (local) or ! (own). The **#link** command uses this to derive the name of any required library files.
%link	Set to the current link list.
%link_arg	Set to its argument by the **#link** command.
%main	Set to the assumed name of the main source file. In make or link mode when not using UNNAMED.PR, this is derived from the project filename, with path and extension removed. Otherwise, it is the supplied source filename complete with path and extension if specified.
%make	Set to on or off by the **#compile**, **#link** and **#dolink** commands, to indicate whether the target file was up to date.

%manual_export	Set this macro on to indicate that the **#link** command should not construct a .LIB file when a DLL is linked. If this macro is not specified, a .LIB file is created automatically from the corresponding .EXP file if found (see Module Definition File below), or from the object files in the link list.
%model	Set by the **#model** command to its first argument, and examined by the #link command.
%name	Set by the **#split** command.
%obj	Set to the object filename in a **#compile** command.
%odir	Set by the **#expand** command.
%opath	Set by the **#expand** command.
%pragmastring	Will always expand to the current state of the #pragma settings - this is useful for debugging.
%prjname	Set to the assumed name of the project - this is usually derived from the project filename, but with the path and extension removed. Where UNNAMED.PR is being used, it is derived from main source filename without source and extension.
%remake	Used within **declare_compiler** macros to determine whether source/object dependencies require a remake.
%remake_jpi	Used within **declare_compiler** macros to determine whether source/object dependencies require a remake. %remake_jpi should be used for object files created by SoftVelocity compilers, which contain additional information.
%reply	Set by the **#prompt** command.
%S	Set by the **#system** command to 32 indicating the instruction set being used to build the project. The **#link** command uses this to derive the name of any required library files.
%src	Set to the source filename in a **#compile** command.
%system	Set by the **#system** command to its first argument, and examined by the **#model** and **#link** commands.
%tail	Set by the **#expand** command.
%tsc	Set to on if a C or C++ source file is compiled.
%tscpp	Set to on if a C++ source file is compiled.
%tsm2	Set to on if a Modula-2 source file is compiled.
%tspas	Set to on if a Pascal source file is compiled.
%warnings	Count of warnings produced by preceding compile or **#file adderrors** command.
%X	The **#link** command uses this to derive the name of any required library files. It adds the letter ‘X’ to the library name, indicating a 32-bit link.

The above macros are examined by the **#link** command to determine which libraries to include, and then set to off.

Basic Compiling and Linking

Compile and link options are specified in a project file by means of the #system, #model and #pragma commands.

#system

#system operating_system [target_type]

The #system command is used to specify the target operating system and file type. The macros %system and %filetype are set to the first and second arguments. See Special Project System Macros below.

The first argument specifies the target operating system, and may be win or win32.

The second argument indicates the target file type, and may be exe, lib, or dll. If omitted, exe is assumed.

The #system command affects the behavior of subsequent #model and #link commands. Therefore a #system command must be specified before either of these. If more than one #system command occurs in a project, each must be followed by a #model command in order to take effect.

#model

#model memory_model [linking_convention]

The #model command is used to specify the memory model to be used for subsequent compiles and links. This memory model will continue to be used until modified by explicit #pragmas, or by another #model command.

The #model command sets the macros %model and %jpicall to its first and second parameters respectively. For example,

```
#model clarion dll
```

is equivalent to

```
#set %model = ‘clarion’
#set %jpicall = ‘dll’
```

The first argument specifies the memory model, which is always ‘clarion’ for Clarion projects. The second indicates the linking convention, which may be dll, lib, or customdll. If omitted, dll is assumed.

Setting the second argument to dll indicates that you will be creating an exe or dll that calls the standard Clarion dlls. Setting the second parameter to lib indicates that you will be creating an exe, lib or dll that includes all the components of the Standard Clarion libraries (and file drivers) in the exe, lib or dll. Using customdll indicates that you are linking to a dll that was previously created with the lib link convention, so the standard Clarion dlls are not linked.

The #system command must be specified before the first #model command.

#pragma

#pragma <#pragma> { , <#pragma> }

The #pragma command modifies the state of the #pragma options which affect the behavior of the SoftVelocity compilers or linker. The syntax and meaning of all #pragmas are discussed under the SoftVelocity #pragmas section below.

The special macro %pragmastring expands to the current state of all #pragma options which are not in their default state - this can be useful for determining exactly which options are being used for a given compile. For example:

```
#message '%pragmastring'
```

Compile and Link Commands

Whenever a file is compiled or linked, the current settings of the compiler or linker options (#pragma settings) are compared to those used when the file was last compiled or linked, to determine whether the file is up to date. If a compile or link is necessary, the current settings are passed on to the compiler or linker.

#compile

#compile<source> [#to <object>] [/ <#pragma> { , <#pragma> }]

{ , <source> [#to <object>] [/ <#pragma> { , <#pragma> }] }

The #compile command causes each nominated source file to be compiled (if necessary). The name of the object file may be specified using #to. If this is omitted, the name is derived from the source filename, with the extension .obj.

Any #pragmas specified in a #compile command apply only to the single source filename that precedes the / character.

The macro %make is set to on if a compile is necessary, off otherwise. The macros %src and %obj are set to the names of the source and object filenames.

Each object file is added to the link list, i.e. there is an implicit:

```
#pragma link( %obj )
```

For example:

```
#compile fred.c #to fred.obj

#compile george.cpp /debug(vid=>full)
```

It is possible to reconfigure the behavior of the Project System when compiling source files of a given extension using the #declare_compiler command. This may also be used to declare actions to perform for different file extensions - for example, to support third-party compilers or preprocessors. See Other Commands below.

#link

#link <target_filename>

The #link command links together (if necessary) all the files in the link list to the nominated executable or library file. The file type is determined by the extension of the nominated target file, or, if there is no extension, by the file type specified in the most recent #system command. If neither are specified, the default is to produce an executable file. The effect of #link is to set the macro %link_arg to the specified filename.

The Project System maintains a list of those files that are to be used as input to the linker the next time an executable or library file is created. This list is known as the *link list*. A filename may be added to the link list using the #pragma link command.

For example:

```
#pragma link (mylib.lib)
```

However, it is seldom necessary to use #pragma link explicitly, as all the SoftVelocity compilers add the resulting object file to the link list whenever a source file is compiled using #compile. In addition, when the #link command is encountered, all required standard library files, and other object files which are imported by those already on the link list are also added to the list. The link list is cleared after each link.

The #link command differs from the similar #dolink command in that (so far as the Project System can determine), any additional object files required are automatically added to the link list before linking. This includes any SoftVelocity library files, and also (with an implicit #autocompile command) all modules imported with IMPORT clauses in SoftVelocity Modula-2 or with #pragma link statements in SoftVelocity C or SoftVelocity C++ source files. In addition, #link will determine from the target file type any additional processing that needs to be applied to the output file.

For certain specialized requirements, the use of #link may be inappropriate—for example, if a specialized startup file is required, or when building library files, where explicit control of exactly which files are included may be preferred. In such cases, the #dolink command should be used.

#dolink <target_filename>

The #dolink command takes the object files which have previously been added to the link list, and combines them into an executable or library file (depending on the extension of the nominated target file), if required to keep the target file up to date. No additional files are added to the link list, so all required files must have been specified previously, by means of #pragma link, #pragma linkfirst, #compile, and #autocompile. For simple projects, the use of #link is preferable because the link list is dynamically maintained by the project system, freeing the developer from this responsibility.

When finished, the #dolink command clears the link list.

See Also:

#pragma link_options (link)

#autocompile

The #autocompile command examines the object files which are currently in the link list, to see which objects they need to be linked with. This would include objects specified using a #pragma link in a SoftVelocity C or C++ source file, or in the case of module based languages such as SoftVelocity Modula-2 imported modules.

Each resulting object file, which is not already in the link list, is then compiled (if necessary) and added to the link list. If there is more than one possible source for a given object file, an error is reported. This process is repeated until the link list stops changing.

It is not necessary to use #autocompile for simple projects where #link is used rather than #dolink, as #link performs an implicit #autocompile.

#ignore

#ignore <filename>

#ignore #pragmastring

There are two forms of the #ignore command. The first, where a filename is specified, tells the Project System to ignore the date of the nominated file when deciding whether or not to compile. This is useful when a ‘safe’ change is made to a widely used header file, to prevent mass recompile.

The special form #ignore #pragmastring directs the Project System to ignore the #pragma settings when deciding whether or not to compile a file. This may be useful, for example, when a new compile-time macro has been defined, but there is no need to recompile everything.

#implib

#implib <libfilename>

The #implib command is used to create (if necessary) a dynamic link library file. There are two forms of this command, which operate in slightly different ways. If a single filename is specified, this names an import library file, which is created (if not up-to-date) from the object files in the link list. The object files are scanned and each public procedure or variable is exported. For example:

```
#pragma link( fred.obj, joe.obj )

#implib mylib.lib
```

In the second form of the #implib command, an import library filename and a module definition file (.exp—see Module Definition File below) are both specified, and the library file is created (if not up-to-date) from the symbols named in the definition file. This form of the command is equivalent to using the tsimplib utility that comes with SoftVelocity C, C++, and Modula-2.

#implib <expfilename> <libfilename>

Using #implib in the second form requires you to create and maintain the list of exports ‘by hand’, whereas the first form exports all public names automatically. The use of a module definition file is an advantage if you need to maintain compatibility with previous versions of an interface, and it also allows you to export only the procedures which need to be exported.

When #implib is used with a module definition file, the link list is cleared.

Conditional Processing and Flow Control

Project file commands may be executed conditionally, using #if, #then, #elsif, #else, and #endif commands. In addition, processing may be stopped with #error and #abort when certain conditions occur.

#if

The syntax of the #if command is as follows :

```
#if <boolean-expression> #then
commands
#elsif <boolean-expression> #then
commands
#else
commands
#endif
```

The #elsif part may be omitted, or may be repeated any number of times. The #else part may be omitted.

The expressions are evaluated in order, until one of them yields true, then the following command sequence is executed. If none of the expressions yield true, and the #else part is present, then the commands following #else are executed. All other commands are ignored.

The syntax and semantics of boolean expressions are described under Boolean Expressions below.

#error

#error <string>

This command terminates the current project. Under the Clarion environment, the Text Editor is opened at the position of the #error command, and displays the supplied string as the error message. For example:

```
#if "%name"="" #then
#error "name not set up"
#endif
```

#abort

#abort [on | off]

This command is used to control whether a failed #compile or #run command will terminate a project. If abort mode is on, a project will be aborted as soon as a #compile fails, or a #run command produces a non-zero return-code. If abort mode is off, a project will only be aborted if an internal command fails, including a #link, #implib or #exemod command.

#abort on will set abort mode to on, while #abort off will turn it off. #abort without one of the above arguments will abort the current project immediately.

The default abort mode is on when running under the Clarion environment.

User Interface

The following commands allow you to collect information and provide feedback during the make process.

#message

#message <string>

This command displays the specified string in the make display window. This can be used to indicate progress through the project file, or to display status messages. For example:

```
#message "finished making %prjname"
```

#prompt

#prompt <promptstring> [<defaultstring>]

This command prompts you to enter a string, by displaying the <promptstring> and waiting for a keyboard entry. The string you enter is returned as the value of the macro %reply. If <defaultstring> is specified, and no keyboard entry is made, the <defaultstring> will be used as the value returned to %reply. For example:

```
#prompt "Command line: " %cline

#set cline = %reply
```

Boolean Expressions

Boolean expressions used in #if and #elsif commands are made up from the following boolean operators (listed in order of precedence):

```
#or
#and
#not
=
#exists
#older
( )
```

#or

boolean-expression = <factor> { #or <factor> }

A boolean expression containing one or more #or operators yields true if the evaluation of any of the factors yields true.

#and

<factor> = <term> { #and <term> }

A factor containing one or more #and operators yields false if the evaluation of any of the terms yields false.

#not

<term> = #not <term>

A term proceeded by the #not operator yields true if the evaluation of the term yields false, and vice versa.

= (comparison)

<term> = string = string

A term containing a comparison operator yields true if the strings are identical, otherwise false. == may be used instead of =.

The = operator and second string may be omitted, in which case the first string is compared against the string "on". That is,

```
DemoSwitch =
```

is equivalent to

```
DemoSwitch = "on"
```

The first string may be replaced by an expression of the form name1(name2), where name2 names a #pragma of class name1. In this case, the expression is replaced by the current setting of the specified #pragma, before the comparison is made.

#exists

<term> = #exists <file-name>

A term containing the #exists operator yields true if the file exists (after applying redirection to the filename), otherwise false.

#older

<term> = <file-name> #older <file-name> { , <file-name> }

A term containing the #older operator yields true if the first file specified is older than at least one of the other files specified, otherwise false. Redirection is applied to all filenames (See The Redirection File below). This operator is often useful to determine whether a post/pre-processing action needs to be performed. For example:

```
#if mydll.lib #older mydll.exp #then

...
```

() Parenthesized Boolean expressions

<term> = (<Boolean-expression>)

A term may consist of a parenthesized Boolean expression, in order to alter or clarify the binding of other Boolean operators. The term yields true if the enclosed Boolean expression yields true. Arbitrarily complex Boolean expressions may be formed.

Filenames and Redirection Analysis

Filenames may be fully qualified, e.g. C:\C70\Orders\Order.tps, in which case, redirection analysis is not done. Alternatively, filenames may not be fully qualified, e.g. ORDERS.TPS, in which case redirection analysis is applied.

Redirection analysis means the project system compares the filename with the *filepatterns* in the current redirection file, until a match is found. Then, the project system searches only those directory paths associated with matching *filepattern* to locate the file.

When creating new files, the project system creates the file in the first directory associated with the matching *filepattern*.

#file Commands

The following file system commands are available:

```
#file adderrors
#file append
#file copy
#file delete
#file move
#file redirect
#file touch
```

#file copy <src-filename> <dst-filename>

This command causes a file to be copied from <src-filename> to <dst-filename>. Both <src-filename> and <dst-filename> must be filenames without wildcard characters. Redirection is applied to both filenames.

#file delete <filename>

This command causes the nominated file to be deleted. <filename> must be a filename without wildcard characters. Redirection is applied to the filename.

#file move <src-filename> <dst-filename>

This command moves (renames) a file from <src-filename> to <dst-filename>. Both filenames must specify files on the same drive. Redirection is applied to both filenames.

#file touch <filename>

This command sets the date and time of <filename> to be the current date and time.

#file append <filename> <string>

This command appends the specified string to <filename>, followed by a CR/LF pair. The file will be created if it does not exist. This command can be used to build log files, etc.

#file redirect [<filename>]

This command changes the current redirection file to <filename>. If no filename is specified, then the command changes the current redirection file to the redirection file that began the project.

At the end of the project file, the redirection file is restored to the redirection file that began the project.

#file adderrors <filename>

This command processes the error messages in the nominated file, and adds them to the errors that will be reported when the project terminates.

Each error message must be in one of the following formats:

```
    (filename lineno,colno): error string
    (filename lineno): error string
    filename(lineno): error string
```

To capture errors from a program with a different error format, a filter program can be used to translate them. For example:

```
    #run ‘masm %f; > %f.err’
    #file adderrors %f.err
    #run ‘myprog %f; | myfilter > %f.err’
    #file adderrors %f.err
```

If any errors are detected, and abort mode is on, the project will terminate and the errors will be reported in the make status window.

The macros %errors and %warnings are set to the number of errors and warnings detected.

Other Commands

#run <commandstring>

This command executes the command specified by <commandstring>. A #run command is generated whenever you add a file to the Programs to execute folder in the Project Tree dialog.

For example:

```
#run "dir > dir.log"
#run "myprog"
```

Note: Filenames within the command string (with the exception of the executable filename itself) are not automatically subject to redirection - #expand may be used before using #run if this is required.

#include <file-name>

A copy of the contents of the nominated file is inserted in the input stream. <filename> should specify a fully qualified filename, or an unqualified filename, in which case redirection analysis is applied (see The Redirection File above).

The current values of the link list, #pragma settings, and macros are fully available to the #include statements. In other words, the #include statements are handled as though they resided within the including .prj file.

#call <file-name>

A copy of the contents of the nominated file is inserted in the input stream. <filename> should specify a fully qualified filename, or an unqualified filename, in which case redirection analysis is applied (see The Redirection File above).

The current values of the link list, #pragma settings, and macros are not available to the #call statements, and the #call statements cannot modify these values in the calling environment. In other words, #call statements are handled as a process that is completely separate from the calling process.

#declare_compiler <file_extension> = <executed_macro>

This defines a macro which is invoked when compiling source files with an extension matching the first parameter. The macros %src and %obj, are set to the names of the source and object files.

Generally, you will not have to use this command explicitly, as all SoftVelocity compilers are pre-declared in the Project System. For example the following is to invoke MASM

```
#declare_compiler asm=
'#set make=%%remake
#if %%make #then
#edit save %%src
#expand %%src
#set _masmsrc=%%opath
#expand %%obj
#set _masmobj=%%cpath
#run "masm %%_masmsrc,%%_masmobj/MX/e; >masmtmp.$$$"
#file adderrors masmtmp.$$$
#file delete masmtmp.$$$
#endif
#pragma link(%%obj)'
```

#rundll <dll_name> <source_filename> <output_filename>

This command invokes an integrated SoftVelocity compiler/utility. The first string is the DLL name, the second is the source filename, and the third is the output filename.

You should never have to use this command explicitly, as all SoftVelocity compilers/utilities are pre-declared in the Project System.

#exemod

#exemod <file-name> <file-name> <file-name>

This command is the equivalent of using the tsexemod utility that comes with SoftVelocity C, C++, and Modula-2. #exemod is required to make advanced overlay model programs, Windows programs and DOS DLLs. However, it is not necessary to use this command explicitly when making Windows programs.

TSEXEMOD is used to modify the header and segment information in a new format executable file (.EXE or .DLL), using the information in a module definition (.EXP) file. For example:

```
      TSEXEMOD binfile.* expfile.exp mapfile.map
```

SoftVelocity #pragmas

All SoftVelocity languages, and the Project System, use a common set of compiler options known as *#pragmas*. In general, pragmas may appear in the source code or in a project file, and the effect will be the same.

Note:

A pragma can be used in the Project language, C++ code, Modula-2 code, or Clarion code. Some only work in certain places. A 'P' to the right of the pragma indicates it can be used in the Project language, a 'C' indicates it can be used in C++ code, a 'M' indicates it can be used in Modula-2 code, and a 'W' indicates that it can be used in Clarion code (CLW files).

Modula-2 Pragma Syntax

Pragmas in SoftVelocity Modula-2 occur in a special form of comment which begins with '(*#'. For example:

```
(*# check( index => off ) *)
```

Old-type Compiler Directives

In the original version of SoftVelocity Modula-2, compiler directives starting with a $ were used to specify compiler options. These directives are still accepted in later versions of SoftVelocity Modula-2, with the following exceptions:

- $B (Ctrl-Break handler). This is no longer supported. Use Lib.EnableBreakCheck instead.
- $D (data segment name). This is supported, but adds the suffix _BSS (for uninitialized data) or _DATA (for initialized data) to the name instead of the D_ prefix.
- $J (use IRET instead of RET). This is not supported. Instead, you should use the pragma:

  ```
  (*# call( interrupt => on ) *)
  ```

- However, you may find that you have to make other changes as well as the effect of the pragma is different from the $J directive:
- $K (C calling convention). This is not supported. Instead, you should use the pragma:

  ```
  (*# call( c_conv => on ) *)
  ```

- $M (code segment name). This is supported but adds the suffix _TEXT to the name instead of the C_ prefix.
- $P (external names for local procedures). This is no longer supported. It is no longer applicable.
- $Q (procedure tracing). This is no longer supported. Instead, you should use the pragma:

  ```
  (*# debug( proc_trace => on ) *)
  ```

- This enables a different method of tracing procedures. Refer to the proc_trace pragma for further details.
- $X (80x87 stack spilling). This is no longer supported (and is no longer necessary).
- $Z (NIL pointer checks). This still does NIL pointer checks but no longer clears memory.
- $@ (preserve DS). This is no longer supported.

The support for these directives has been included with later systems so that your old programs and modules will recompile with minimum changes. However, you should avoid using the old directives with new programs, and use pragmas instead.

C and C++ Pragma Syntax

Pragmas are an integral part of the C and C++ languages, and are implemented as compiler directives:

```
#pragma check( index => off )
```

Project System Pragma Syntax

Pragmas in the Project System use a similar syntax to the C and C++ languages:

```
#pragma check(index => off)
```

Pragmas in the Project System may also be specified in the #compile command, to apply to a single compilation. For example:

```
#compile mandel.mod /debug(vid=>on)
```

Pragma Classes

A #pragma takes the form #pragma class(name=>value). The #pragma classes are as follows:

Call #pragmas

Check #pragmas

Code #pragmas

Data #pragmas

Debug #pragmas

Define #pragmas

Link and Linkfirst #pragmas

Link_option #pragmas

Module #pragmas

Name #pragmas

Optimize #pragmas

Option #pragmas

Project #pragmas

Save and Restore #pragmas

Warn #pragmas

Call #pragmas

#pragmas with the class name *call* affect all aspects of calling conventions, code segments, and code pointers. The current settings of the call #pragmas at the point at which a procedure's definition is encountered, determines the calling convention that is used to call the procedure. SoftVelocity compilers detect if an inconsistent calling convention is used when a procedure is called. The type-safe linker reports an error if the calling conventions attributed to a given procedure do not match in every object file.

The following call #pragmas are available:

```
#pragma call(c_conv => on | off)

#pragma call(ds_entry => identifier)

#pragma call(ds_eq_ss => on | off)

#pragma call(inline => on | off)

#pragma call(inline_max => Number)

#pragma call(near_call => on | off)

#pragma call(o_a_copy => on | off)

#pragma call(o_a_size => on | off)

#pragma call(opt_var_arg => on | off)

#pragma call(reg_param => RegList)

#pragma call(reg_return => RegList)

#pragma call(reg_saved => RegList)

#pragma call(result_optional => on | off)

#pragma call(same_ds => on | off)

#pragma call(seg_name => identifier)

#pragma call(set_jmp => on | off)

#pragma call(standard_conv => on | off)

#pragma call(var_arg => on | off)
```

Note:

A pragma can be used in the Project language, C++ code, Modula-2 code, or Clarion code. Some only work in certain places. A 'P' to the right of the pragma indicates it can be used in the Project language, a 'C' indicates it can be used in C++ code, a 'M' indicates it can be used in Modula-2 code, and a 'W' indicates that it can be used in Clarion code (CLW files).

#pragma call(near_call => on | off)   cpm

Specifies whether procedure calls are near or far. When on, the compiler calls procedures with near calls. The compiler can only use near calls if the calling and called procedures are in the same segment. The compiler checks that this is the case.

The default value is off. This example forces near calls:

```
#pragma call(near_call=>on)
```

#pragma call(same_ds => on | off)   cpm

Specifies whether to load the data segment (DS) register on entry to a procedure. When on, DS will not be loaded as part of the procedure prolog. This will only be correct when the DS setting of the calling procedure matches that of the called procedure. The compiler checks that this is the case.

This option is off by default. For example:

```
#pragma call(same_ds => on)
```

This stops DS from being loaded in the procedure prolog.

#pragma call(c_conv => on | off)   cpm

When on, this option enables the Microsoft C calling convention. In this convention, the compiler pushes procedure parameters in right to left order on the stack and the caller pops these parameters off the stack.

This is not the default, so you should only use this #pragma when interfacing to Microsoft C code. For example:

```
#pragma call(c_conv=>on)
```

You can also use the cdecl keyword in C and C++ to achieve the same effect.

See also the standard_conv #pragma, which has the same effect for C and C++, but is ignored for Modula-2. The standard_conv #pragma is set off by default.

#pragma call(inline => on | off)   cm

If this #pragma is set on before a procedure definition, the compiler makes a copy of the procedure in the code rather than using a call instruction. The default value is off.

You can use this convention for any procedure, but this #pragma is mainly used together with the reg_param #pragma for simple machine-code procedures. For example:

```
#pragma save
#pragma call(inline => on, reg_param => (dx,ax))
static void outp(int port, unsigned char byt)=
{
0xEE, /* out dx,al */
};
#pragma restore
```

makes outp an inline procedure, so a call to it appears as a single 80x86 machine instruction: out dx,al.

#pragma call(seg_name => identifier)   cpm

Specifies the code segment name. call(seg_name => nnn) means that the compiler places the code for the procedure in segment nnn_TEXT. The default value depends on the memory model. In the small and compact models, the default is null; in the other models it is the name of the source file. For example, a code segment named _TEXT would be specified as:

```
#pragma call(seg_name => null)
```

and a code segment named MYCODE_TEXT, would be specified as:

```
#pragma call(seg_name => MYCODE)
```

The default setting is language dependant, and is not defined by the Project System.

#pragma call(ds_entry => identifier) cpm

This #pragma indicates a segment name which the DS register will point to throughout the execution of a procedure. If the identifier is null, the compiler names the segment _DATA. If the identifier is none, the compiler does not assume a fixed DS during procedure execution and uses DS as a general purpose segment register like ES.

```
#pragma call(ds_entry => MYDATA)
```

This example indicates that on entry to the procedure, DS will point to the segment named MYDATA_DGROUP.

#pragma call(reg_param => RegList) cm

SoftVelocity languages pass procedure parameters in machine registers rather than using the stack. This generates smaller and faster code. This #pragma allows you to fine-tune individual procedure calls for maximum speed. Other vendors' languages use a less efficient calling convention; you must, therefore, disable this calling convention when interfacing to precompiled objects written for these compilers (see the Advanced Programmer's Guide that comes with SoftVelocity C, C++, and Modula-2, Chapter 5: Multi-language Programming). This #pragma has no effect on structure parameters, which are always passed on the stack.

The argument for reg_param is a register list, specifying which registers should be used. Registers for parameters are allocated left to right from the list. The table shows how the compiler allocates registers dependent on parameter types:

```
1 byte ax, bx, cx, dx

2 bytes ax, bx, cx, dx, si, di

4 bytes ax, bx, cx, dx, si, di for low word.

ax, bx, cx, dx, si, di, es, ds for high word.

floating point st0, st1, st2, st3, st4, st5, st6
```

Note that the es and ds registers will only be used for the high word of a 4-byte parameter where that parameter is of pointer type. If either the low or high word cannot be allocated, then the whole parameter is passed on the stack.

When the compiler exhausts the list of registers, it passes the parameter on the stack. If you specify an empty list, the compiler uses the stack for all parameters.

The default setting for the SoftVelocity calling convention is:

```
#pragma call(reg_param=>(ax,bx,cx,dx,st0,st6,st5,st4,st3))
```

The default setting for the stack calling convention is:

```
#pragma call(reg_param => ())
```

#pragma call(reg_saved => RegList) cm

This #pragma specifies which registers a procedure preserves. The argument RegList is a list that specifies the set of registers.

The default set for the SoftVelocity calling convention is:

```
ccall(reg_saved=>(ax,bx,cx,dx,si,di,ds,st1,st2))
```

The default set for the stack calling convention is:

```
#pragma call(reg_saved=>(si,di,ds,st1,st2))
```

#pragma call(o_a_size => on | off)   m

When on, this option passes the size of open array parameters on the stack:

```
(*# call( o_a_size => on ) *)
```

This #pragma has no effect for value parameter open arrays, unless the o_a_copy #pragma is set off.

The default setting is on.

#pragma call(o_a_copy => on | off)   m

When on, open array parameters are copied onto the stack as part of the procedure prolog. If off, only a reference to the array is passed. Note that the open array parameters size must be passed in order for a copy to be made - see #pragma call(o_a_size). The default setting is on.

#pragma call(ds_eq_ss => on | off)   m

It controls whether VAR parameters use 16- or 32-bit pointers. The default setting is on for small and medium models, otherwise off.

#pragma call(var_arg => on | off)   m

When on, it implies that the following procedures take a variable number of arguments. This effectively disables the "too many arguments" error that the compiler would normally detect. The consequence however, is that the compiler cannot carry out any type checking on the arguments.

This #pragma should be used when calling C procedures (such as printf) where the number of arguments varies:

```
(*# call(var_arg => on,
reg_param=>(),
c_conv=>on ) *)
```

The default setting is off.

#pragma call(reg_return => RegList)   cm

This #pragma is used to specify the registers to be used for return values of integer, pointer and floating point types. For example:

```
#pragma call(reg_return => (bx,cx))
```

The default setting is:

```
#pragma call(reg_return=>(ax,dx,st0))
```

#pragma call(result_optional => on | off)   m

It can be used to call a procedure as a proper procedure without generating a compiler error. For example:

```
(*# save *)
(*# module( result_optional => on ) *)
PROCEDURE FuncProc(x: CHAR): CARDINAL;
(*# restore *)
```

With this declaration, you can write both of the following:

```
    i := FuncProc('a');
    FuncProc('a');
```

This is only useful when the called procedure has a side effect that is more important than the result. It is particularly useful when calling SoftVelocity C library procedures.

The default setting is off.

#pragma call(set_jmp => on | off)   cm

This #pragma should only be used for the library routines which implement non-local jumps. The effect is to inform the compiler of the non-standard register saving properties of these routines.

#pragma call(inline_max => Number)   cpm

This #pragma controls the largest procedure which is inlined. The default setting is 12, which corresponds to the minimum code size for most programs. A larger value increases the code size and may accelerate code execution.

The #pragma takes effect for each call, so a procedure may be called in different ways at different places.

Note: Procedures are not inlined if the body has not been compiled before the call.

#pragma call(standard_conv => on | off)   c

The effect on C and C++ programs is the same as the call(c_conv) #pragma. For Modula-2 there is no effect. The default is off.

#pragma call(opt_var_arg => on | off)   cp

This #pragma controls whether optimized entry sequences are generated for procedures with variable parameter lists. The default is on.

Check #pragmas

#pragmas with the class name *check* control run-time error checking. These can help you to locate erroneous program logic, but at the expense of slower execution. All these #pragmas default to off.

When a run-time check detects an error, the default action is to terminate the process and create the file CWLOG.TXT.

The following check #pragmas are available:

```
#pragma check(guard => on | off)
#pragma check(index => on | off)
#pragma check(nil_ptr => on | off)
#pragma check(overflow => on | off)
#pragma check(range => on | off)
#pragma check(stack => on | off)
```

Note:

A pragma can be used in the Project language, C++ code, Modula-2 code, or Clarion code. Some only work in certain places. A 'P' to the right of the pragma indicates it can be used in the Project language, a 'C' indicates it can be used in C++ code, a 'M' indicates it can be used in Modula-2 code, and a 'W' indicates that it can be used in Clarion code (CLW files).

#pragma check(stack => on | off) cpmw

When on, the run-time system checks that your program does not run out of stack space. You can increase the size of the stack using the data(stack_size) #pragma.

#pragma check(nil_ptr => on | off) cpmw

When on, the run-time system checks for any dereference of NULL or NIL pointers.

#pragma check(range => on | off) pm

When on, a range check is performed whenever a value is assigned to a variable of subrange or enumerated type. In addition, compile-time values are checked to ensure that they are in the range of their type.

#pragma check(overflow => on | off) pm

When on, the run-time system checks that numeric values do not go out of range.

#pragma check(index => on | off) cpmw

When on, the run-time system checks for the use of an array index larger than the array size.

#pragma check(guard => on | off) pm

This #pragma controls whether checks are performed on the checked-guard operator.

Code #pragmas

#pragmas with the class name *code* control internal code generation optimizations.

The following code #pragmas are available:

```
#pragma code(opt_var_arg => on | off)
#pragma code(standard_float => on | off)
```

Note:

A pragma can be used in the Project language, C++ code, Modula-2 code, or Clarion code. Some only work in certain places. A 'P' to the right of the pragma indicates it can be used in the Project language, a 'C' indicates it can be used in C++ code, a 'M' indicates it can be used in Modula-2 code, and a 'W' indicates that it can be used in Clarion code (CLW files).

#pragma code(opt_var_arg => on | off)   w

If value of this pragma is "on" (default value) the compiler performs additional optimization for the procedure stack frame for variable parameters. This pragma must always be set to "on".

#pragma code(standard_float => on | off)   w

If value of this pragma is "off" (default), the compiler generates the code to save some FPU registers (the exact list is dependent on values of other pragmas, procedure type (normal or ROUTINE), etc.) on the entry to procedure and restore them on exit, if the procedure is using them. Otherwise, values of FPU registers are not saved in the procedure.

Data #pragmas

#pragmas with the class name *data* affect data segmentation, data pointers and all aspects of data layout. The current settings of the data #pragmas at the point of a variable's declaration will affect the way in which it is accessed.

The following data #pragmas are available:

```
#pragma data(c_far_ext => on | off)
#pragma data(class_hierarchy => on | off )
#pragma data(compatible_class => on | off)
#pragma data(const_assign => on | off )
#pragma data(const_in_code => on | off)
#pragma data(cpp_compatible_class => on | off )
#pragma data(ds_dynamic => on | off )
#pragma data(ext_record => on | off )
#pragma data(far_ext => on | off )
#pragma data(heap_size => Number)
#pragma data(near_ptr => on | off)
#pragma data(packed => on | off )
#pragma data(seg_name => identifier)
#pragma data(stack_size => Number)
#pragma data(threshold => Number)
#pragma data(var_enum_size => on | off)
#pragma data(volatile => on | off)
#pragma data(volatile_variant => on | off )
```

Note:

A pragma can be used in the Project language, C++ code, Modula-2 code, or Clarion code. Some only work in certain places. A 'P' to the right of the pragma indicates it can be used in the Project language, a 'C' indicates it can be used in C++ code, a 'M' indicates it can be used in Modula-2 code, and a 'W' indicates that it can be used in Clarion code (CLW files).

#pragma data(seg_name => identifier)   cpmw

The #pragma data(seg_3name=>xxx) specifies that the compiler should place global initialized data objects in a segment named xxx_DATA, and global uninitialized data objects in a segment named xxx_BSS. These both have group name xxx and are in the FAR_DATA class. If the size of a data object is larger than the global data threshold, the compiler places the object in a separate segment.

The following example makes the names of the default segments: MYDATA_DATA and MYDATA_BSS. These segments are in group MYDATA and have class FAR_DATA:

```
#pragma data(seg_name => MYDATA)
```

You can also specify null, to indicate the names _BSS and _DATA. The default value is null in all models except for extra large and multi-thread. For example:

```
#pragma data(seg_name => null)
```

#pragma data(far_ext => on | off)   cpw

When on, the code generator does not assume that external variables are in the segment specified by the segment #pragma. The #pragma defaults to on. For example:

```
#pragma(seg_name=>MYDATA, far_ext=>off)
```

makes the name of the default segments MYDATA_DATA and MYDATA_BSS in group MYDATA. The compiler assumes external data objects to be in the same segment.

#pragma data(c_far_ext => on | off)   pm

When on, the code generator does not assume that external variables are in the segment specified by the seg_name #pragma. The #pragma defaults to off in all memory models. For example:

```
(*# data(seg_name => MYDATA, c_far_ext => off ) *)
```

makes the name of the default segments MYDATA_DATA and MYDATA_BSS in group MYDATA. This #pragma is not particularly useful in Modula-2 except for interfacing to C.

#pragma data(near_ptr => on | off)   cm

Specifies whether data pointers are near or far. This #pragma also affects pointers generated by the & operator and by implicit array to pointer conversions in C and C++. For example:

```
#pragma data(near_ptr => on)
```

makes data pointers near.

#pragma data(volatile => on | off)   m

Variables declared when this #pragma is set to on are considered to be volatile, and will always be kept in memory, rather than being kept in registers across statements.

The default setting is off. This #pragma is not allowed in a project file, and is not available for C and C++, where the volatile keyword should be used.

#pragma data(volatile_variant => on | off)   m

The effect is as for #pragma data(volatile), but applies to variables of variant record types only. The default setting is off.

#pragma data(ext_record => on | off)   pm

Normally SoftVelocity does not allow two fields in different alternatives of a variant record to have the same name. Using this #pragma:

```
(*# data( ext_record => on) *)
```

will allow you to use the same name in different alternatives, if the fields are located at the same offset in the variant record and they have the same data type.

The default setting is off.

#pragma data(var_enum_size => on | off)   pm

Enumeration constants with less than 256 alternative values are normally stored in one byte. Switching this option off:

```
(*# data( var_enum_size => off) *)
```

will force the compiler to store them as two-byte quantities. This is particularly useful for interfacing to third-party libraries and operating system calls that expect a word value. Without this #pragma the enumeration would be byte rather than word size.

The default setting is on.

#pragma data(stack_size => Number)   cmw

Specifies the size of the stack. You must place this #pragma in the file containing the main procedure (the main module in Modula-2). If the stack size cannot be set to the specified size, the compiler uses the largest possible size. The default size is 16K bytes. For example:

```
#pragma data(stack_size => 0x6000)
main()
{
 /* statements */
}
```

makes the size of the run-time stack 0x6000 bytes (24K).

#pragma data(packed => on | off)   pm

This #pragma controls whether record fields are packed at bit level. The default setting is off.

#pragma data(const_in_code => on | off)   pw

This #pragma controls whether constants are put in to a code or data segment. The default setting is on.

#pragma data(class_hierarchy => on | off)   pm

This #pragma controls whether information about class hierarchies is included in the class descriptor (method table). The information is used by the IS operator and TypeGuards with check on. The default setting is on.

#pragma data(cpp_compatible_class => on | off) pm

This #pragma controls whether the compiler includes extra information in class descriptors to provide compatibility with C. The default setting is off.

#pragma data(compatible_class => on | off)   cp

This #pragma controls whether the compiler includes the correct information in class descriptors to provide compatibility with Modula-2. The default setting is off.

#pragma data(threshold => Number)   cpmw

This #pragma sets the global data threshold. This determines at what size a data object is placed in a segment of its own. The default setting is 10000 bytes.

#pragma data(const_assign => on | off)   pm

This #pragma controls whether it is possible to assign to a structured constant. The default setting is off.

Note: If const_in_code=>on is specified, assignments to constants will result in protection violations.

Debug #pragmas

#pragmas with the class name *debug* control the amount of additional information produced by the code-generator to assist debugging programs.

The following debug #pragmas are available.

```
#pragma debug(line_num => on | off)

#pragma debug(proc_trace => on | off)

#pragma debug(public => on | off)

#pragma debug(vid => off | min | full)
```

Note:

A pragma can be used in the Project language, C++ code, Modula-2 code, or Clarion code. Some only work in certain places. A 'P' to the right of the pragma indicates it can be used in the Project language, a 'C' indicates it can be used in C++ code, a 'M' indicates it can be used in Modula-2 code, and a 'W' indicates that it can be used in Clarion code (CLW files).

#pragma debug(vid => off | min | full)   cpmw

When full, the compiler places information for the SoftVelocity Visual Interactive Debugger (VID) into a .DBD file. Use this option when debugging your program with the SoftVelocity debugger.

Note: This #pragma disables the register usage and stack frame optimizations, allowing full access to variables within the debugger. All local variables are treated as volatile, to ensure that their values are not held in registers across statements, thus ensuring that the debugger can access their values at all times.

When min, the compiler performs the optimizations described above, and does not treat local variables as volatile. The debugger can still be used, but cannot reference local variables and some stack frames.

When off, the compiler generates no debugger information, thus speeding compile, generating the best possible code, and saving disk space. The default setting is off.

#pragma debug(proc_trace => on | off)   pmw

When this #pragma is on, the compiler generates instructions to call the procedures EnterProc and ExitProc on, respectively, entering and exiting every procedure. These procedures can then perform any procedure tracing you may require.

Note: You should ensure that this #pragma is off for the EnterProc and ExitProc procedures themselves, otherwise infinite recursion will occur and your program will undoubtedly crash.

The two procedures must be visible to the module in which proc_trace is set on. This means that the module itself must define the procedures ExitProc and EnterProc or the module must specifically import them using an unqualified import.

The default setting is off.

#pragma debug(line_num => on | off)   cpmw

This #pragma causes the compiler to generate line number information for debuggers such as symdeb. This information is stored in object files and printed in the map file. The default setting is off.

#pragma debug(public => on | off)   cpm

This causes private objects to be made public to facilitate the use of debuggers such as symdeb. It may cause duplicated public warnings at link-time in languages such as C and C++ which do not have a modular structure. These warnings may be safely ignored, although it is recommended that such procedures should be renamed to avoid possible confusion. The default setting is off.

Define #pragmas

A #pragma whose class name is *define* is used to define a conditional compile symbol for subsequent compiles. The symbol is available for interrogation by the OMIT and COMPILE compiler directives. See the Language Reference for more information. This #pragma may only be used in project files.

A define #pragma takes the form:

```
#pragma define(ident=>value)
```

where ident names the symbol and value specifies the value it is given.

For Modula-2, the given identifier is defined as a boolean constant with value TRUE if the value on was specified, otherwise FALSE. For C and C++, the given identifier is defined as a macro. If the value on is specified, the macro is defined to the value 1. If the value off is specified, the macro is not defined. Any other value will cause the identifier to be defined as a macro expanding to the given value. Only a single C or C++ token may be specified, or the compiler will report an error. To define a macro where the value is a string literal, use a command of the form #pragma define (name => '"fred"').

The following define #pragmas are available:

```
#pragma define(asserts => 0 | 1 )

#pragma define(bcd_arithmetic => on | off )
#pragma define(bcd_large => on | off )
#pragma define(bcd_ulong => on | off )
#pragma define(big_code => n)

#pragma define(compatible_modulus=> on | off )
#pragma define(dll_mode => 0 | 1 )
#pragma define(init_priority => n)
#pragma define(local_implicits => 0 | 1 )
#pragma define(logical_round => on | off )
#pragma define(maincode => on | off )
#pragma define(profile => on | off )
#pragma define(safe_heap => 0 | 1 )
#pragma define(stack_threshold => size )
#pragma define(wdnu => off | on)
#pragma define(wpnu => off | on)
#pragma define(zero_divide => on | off )
```

Note:

A pragma can be used in the Project language, C++ code, Modula-2 code, or Clarion code. Some only work in certain places. A 'P' to the right of the pragma indicates it can be used in the Project language, a 'C' indicates it can be used in C++ code, a 'M' indicates it can be used in Modula-2 code, and a 'W' indicates that it can be used in Clarion code (CLW files).

#pragma define(asserts => 0 | 1)   pw

If the value is 0, and the debug information is not active (generating), the compiler generates nothing for any ASSERT statement detected. If the value is 1, or the debug information is being generated, the compiler produces the code to show the assert dialog. This code can be replaced by setting SYSTEM{PROP:AssertHook} property. If the pragma is set to 1, the call to show the assert dialog is generated *regardless* of the value of the *debug(vid)* pragma.

#pragma define(bcd_arithmetic => on | off)   pw

Specifies use of Binary Coded Decimal (BCD) arithmetic when on (default) and forces use of Floating Point arithmetic when off.

#pragma define(bcd_copies => 0 | 1)   pw

If pragma is set to 1 (default value), the compiler is trying to keep expression type as DECIMAL if all arguments are integers or DECIMALs or PDECIMALs as long as possible. Otherwise, arguments are converted to REALs.

#pragma define(bcd_large => on | off)   pw

Enables or disables use of DECIMAL and PDECIMAL variables greater than fifteen (15) digits. The default is on (enabled).

#pragma define(bcd_ulong => on | off)   pw

Specifies use of Binary Coded Decimal (BCD) arithmetic for ULONG variables when on (default) and forces use of Floating Point when off. This define is overridden with define(bcd_arithmetic=>off)

#pragma define(big_code => n)   p

Specifies number of procedures per segment. By default (n = 0) all procedures ina single module go into a single code segment. Setting the value of n for a specific module "breaks up" the module containing a large number of procedures which "breaks" the 64K code segment limit.

#pragma define(compatible_modulus=> on | off)   p

Controls the sign of the result of modulus division. If the pragma is set to zero (OFF), the modulus result uses the sign of the divisor. If the pragma is set to one (ON), the modulus result uses the sign of the dividend.The default is off.

Example:

If compatible_modulus = 0, then

42 % -5 = -2 (-5 is the divisor(-))

If compatible_modulus = 1, then

42 % -5 = 2 (42 is dividend (+))

#pragma define(dll_mode => 0 | 1) pw

The project system sets this pragma to 0 if the link mode is set to *Local* in the project, and sets it to 1 otherwise. The program should not override the project settings.

#pragma define(init_priority => n) pw

Specifies a number (n) that is compatible with the C++module priority schema. Default is 5.

#pragma define(local_implicits => 0 | 1) pw

If the pragma is set to 0 (default value), implicit variables that are declared in the procedure are accessible from ROUTINEs and methods of local classes. If pragma is set to 1, implicit variables with the same name are allocating separately in each scope.

#pragma define(logical_round => on | off) pw

Specifies rounding behavior when truncating a REAL to a LONG. When on, the result is rounded up if the REAL value is "close to" the next larger integer. When off (default), no rounding occurs.

#pragma define(maincode => on | off) pw

Enables (on—the default) or disables (off) generation of initialization code. Turn maincode off when compiling generic modules or LIB modules.

#pragma define(profile => on | off) pw

Specifies the compiler will invoke a procedure call at the beginning and end of compiling each procedure. This allows you to implement your own profiler. The prototypes for these procedures must be:

```
EnterProc(UNSIGNED Line,*CSTRING Proc,*CSTRING File),NAME('Profile:EnterProc')
LeaveProc(),NAME('Profile:LeaveProc')
```

The EnterProc is called at the beginning of each procedure and LeaveProc at the end.

#pragma define(safe_heap => 0 | 1) pw

If the pragma is set to 1 (default value), the compiler generates the code to set a pointer to NULL after it has been freed. Otherwise, the value of the pointer remains the same as it was before the call to free().

#pragma define(stack_threshold => size) pw

Specifies the size (in bytes) of the threshold at which any data structure larger than the specified size (default is 16384) is assigned heap memory instead of stack memory.

#pragma define(wpnu => on | off) pw

Parameter never used in function. When on, the compiler checks for a parameter that the code never uses, so declaration of dummy parameters generates warnings. The default setting is off.

#pragma define(wdnu => on | off) pw

Variable declared but never used. When on, the compiler checks whether a local variable has been declared but never used in the function. The default setting is *off*.

#pragma define(zero_divide => on | off) pw

Specifies divide by zero behavior. When on, division by zero returns zero. When off (the default), division by zero returns an exception.

Link #pragmas

```
#pragma link( <filename> {,<filename> } )
#pragma linkfirst( <filename> )
```

These #pragmas may be specified in a project file, in which case the nominated files are added immediately to the link list. In addition, the link #pragma may be specified in a C or C++ source file, in which case the nominated files will be added to the link list when an autocompile command is executed in the Project System, if any files already on the link list had this #pragma specified. For example:

```
#pragma link( file1.obj, file2.obj, file3.lib )
#pragma linkfirst (initexe.obj)
```

If no extension is given .obj is assumed. Files specified using #pragma link are added to the end of the link list (unless already present). A file specified using #pragma linkfirst is linked before the link list. Only one file may be specified for each link using #pragma linkfirst.

Link_Option #pragmas

#pragmas with the class name *link_option* are used to specify linker options. These #pragmas may only occur in project files.

The following link_option #pragmas are available:

```
#pragma link_option(case => on | off )
#pragma link_option(decode => on | off )
#pragma link_option(link=> <string>)
#pragma link_option(map => on | off)
#pragma link_option(overlay => on | off )
#pragma link_option(pack => on | off )
#pragma link_option(shift => num)
#pragma link_option(share_const => on | off)
#pragma link_option(icon => iconname)
```

Note:

A pragma can be used in the Project language, C++ code, Modula-2 code, or Clarion code. Some only work in certain places. A 'P' to the right of the pragma indicates it can be used in the Project language, a 'C' indicates it can be used in C++ code, a 'M' indicates it can be used in Modula-2 code, and a 'W' indicates that it can be used in Clarion code (CLW files).

#pragma link_option(map => on | off) p

Controls whether a map file is generated with information about segment sizes and publics etc. The default is to create a map file.

#pragma link_option(case => on | off) p

This #pragma controls whether the linker treats upper/lower case as significant when linking. The default is case=>off.

#pragma link_option(pack => on | off) p

This #pragma controls whether segments are packed together. The default is pack=>on.

#pragma link_option(decode => on | off) p

This indicates whether the linker should produce decoded names in the MAP file, as well as their public symbols. The option is set to on if any C source files are included in a project, otherwise off.

#pragma link_option(shift => num) p

This specifies the segment alignment shift count for new-format executables. The default is 4, indicating that segments are aligned on 16-byte boundaries.

#pragma link_option(link => <string>)   p

This specifies the project system command to execute on #dolink.

#pragma link_option(share_const=> on | off) p

This pragma controls whether the 16-bit linker commons-up identical constants. The default is on, making the exe file smaller, but C programmers may want to turn it off if they are relying on constants having different addresses.

#pragma link_option(icon => iconname)   p

This pragma specifies the name of the application icon file (icon => MyIcon.ico).

Module #pragmas

#pragmas with the class name *module* control options that apply to an entire source file or module. These #pragmas should be specified at the top of any source files to which they apply, or in the project file.

The following module #pragmas are available:

```
#pragma module(implementation => on | off )
#pragma module(init_code => on | off )
#pragma module(init_prio => Number)
#pragma module(smart_link => on | off)
```

Note:

A pragma can be used in the Project language, C++ code, Modula-2 code, or Clarion code. Some only work in certain places. A 'P' to the right of the pragma indicates it can be used in the Project language, a 'C' indicates it can be used in C++ code, a 'M' indicates it can be used in Modula-2 code, and a 'W' indicates that it can be used in Clarion code (CLW files).

#pragma module(init_code => on | off) pm

When on, it implies that the module contains initialization code to be run when the program is loaded and before the main module is executed. Switching the option off is useful for modules written in other languages, as it will stop the linker warning of undefined symbols:

```
(*# module( init_code => off ) *)
```

Note: If an implementation module sets this #pragma off, then there is a knock-on effect, i.e., all imported modules must also have init_code set to off.

The default setting is on.

#pragma module(implementation => on | off) pm

This #pragma specifies whether or not a definition file (.DEF or .ITF) has a corresponding object file. It should be turned off if the definition file defines interfaces to routines in a different language, to prevent the Project System from attempting to remake the corresponding object file. The default is on.

This #pragma can also be used in the implementation part of a module, before any module source code. In this case it overrides the default naming of the associated object file. Normally the name of the .OBJ file corresponding to a module is taken from the module name. When this #pragma is set off, the object filename will be taken from the filename, not the module name.

#pragma module(smart_link => on | off) cpm

Setting this #pragma to off disables the smart linking feature, to the extent that either all or none of the objects in each segment from a compile will be included in a link. This may result in quicker linking, and also may allow other linkers (such as Microsoft) to be used. (There are many potential problems with trying to use a non-SoftVelocity linker, and it is definitely not recommended). The default setting is on.

#pragma module(init_prio => Number)   cpw

This #pragma is available under C only. It defines a priority for the initialization code for static objects. Normally the initialization order is undefined between files, but this #pragma allows you to control the initialization order in that files with higher priority are initialized before modules with lower priority. The number must be a value between 0 and 32. The default value is 16, and the C library uses values between 25 and 32 (it is therefore not recommended to use values in this range, otherwise part of the library may not have initialized before user code is executed).

Name #pragmas

#pragmas with the class *name* control aspects of linkage naming. However, the C programmer should also be familiar with C name mangling and extern declarations.

The following name #pragmas are available:

```
#pragma name(prefix => (none | modula | c | os2_lib | windows))

#pragma name(prefix => string)

#pragma name(upper_case => on | off)
```

Note:

A pragma can be used in the Project language, C++ code, Modula-2 code, or Clarion code. Some only work in certain places. A 'P' to the right of the pragma indicates it can be used in the Project language, a 'C' indicates it can be used in C++ code, a 'M' indicates it can be used in Modula-2 code, and a 'W' indicates that it can be used in Clarion code (CLW files).

#pragma name(upper_case => on | off)   cp

This #pragma is available in C and C++ Only. It specifies whether public names should be converted to upper case. You would use this when interfacing to Pascal, or to third party C libraries. The default setting is off.

#pragma name(prefix)   cpm

There are two forms of this #pragma:

In Modula-2:	**name(prefix => (none | modula | c | os2_lib | windows))**
In C and C++:	**name(prefix => string)**

#pragma name(prefix => (none | modula | c | os2_lib | windows)) mp

This #pragma is available under Modula-2 (under C and C++ the syntax is slightly different - see #pragma name(prefix => string).

The name(prefix) #pragma specifies the prefix and case of the public names that the compiler uses. The public names are names for non-static procedures and external data objects. By default, SoftVelocity Modula-2 prefixes all external names with the name of the module followed by an '@' for data and a '$' for procedures. You will need to use this #pragma to interface to SoftVelocity C.

The prefix #pragma specifies which prefix scheme to use:

Modula	Use the SoftVelocity Modula-2 naming convention of prefixing all external names with the name of the module and an '@' or a '$'.
none	Puts no prefix on external names.
c	Use the C naming convention (adds an underbar, '_' to all external names).
os2_lib	Use the OS/2 library standard (prefix all external names with the module name).
window	Use the Microsoft Windows external naming convention.

s

#pragma name(prefix => string) cp

This #pragma is available under C and C++ Only. Under Modula-2 the syntax is slightly different - see #pragma name(prefix => (none | modula | c | os2_lib | windows)).

The value is a string specifying the prefix to all public names. An empty string specifies no prefix. The default prefix is an underbar.

If you wish to interface to SoftVelocity Modula-2, you can use this #pragma to specify the module prefix with a dollar ($) suffix. For example, to use the WrStr procedure from module IO:

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

void WrCard(unsigned);
```

In C, a Pascal or Modula2 linkage specification can specify a module name within the linkage specification, in which case the use of this #pragma is not necessary.

The default setting is language-dependent. The Project System does not set a default value for this macro.

Optimize #pragmas

#pragmas with the class name *optimize* control optimizations performed by the SoftVelocity code generator. By default, all optimizations are enabled. Turning off an optimization will result in poorer code quality, and is unlikely to have a significant impact on compile times.

The following optimize #pragmas are available:

```
#pragma optimize(alias => on | off)
#pragma optimize(const => on | off)
#pragma optimize(copro => 87 | 287 | 387)
#pragma optimize(cpu => 86 | 286 | 386 | 486)
#pragma optimize(cse => on | off)
#pragma optimize(jump => on | off)
#pragma optimize(loop => on | off)
#pragma optimize(peep_hole => on | off)
#pragma optimize(regass => on | off)
#pragma optimize(speed => on | off)
#pragma optimize(stk_frame => on | off)
```

Note:

A pragma can be used in the Project language, C++ code, Modula-2 code, or Clarion code. Some only work in certain places. A 'P' to the right of the pragma indicates it can be used in the Project language, a 'C' indicates it can be used in C++ code, a 'M' indicates it can be used in Modula-2 code, and a 'W' indicates that it can be used in Clarion code (CLW files).

#pragma optimize(cse => on | off)   cpm

When on, the compiler minimizes evaluation of complete expressions by keeping partial results in a temporary register. The default setting is on.

#pragma optimize(const => on | off)   cpm

When on, the compiler will hold frequently used constants in registers to produce faster code. The default setting is on.

#pragma optimize(speed => on | off)   cpmw

When on, SoftVelocity tries to make the code run as fast as possible without regard for the code size. When off, SoftVelocity tries to make the code as small as possible.

A good example of the difference between optimizing for speed and optimizing for space is the use of a for loop. When optimizing for speed, the compiler might use nop instructions to place jump labels inside the for loop on even boundaries. The 80x86 architecture makes this much quicker than odd boundaries, but each nop adds another byte to the code size. This means that when optimizing for space, the compiler eliminates the extra nop instructions. The default setting is on.

#pragma optimize(stk_frame => on | off)   cpm

When on, the compiler will only make stack frames where required, thus eliminating the need to set up the BP register. This optimization can only be made when all parameters and local variables for a procedure can be held in machine registers.

When off, the compiler always sets up the BP register, thus allowing a complete activation stack listing while debugging. The default setting is on.

#pragma optimize(regass => on | off)   cpm

When on, the compiler spends time finding the best allocation of registers for variables. This results in fast and tight code but slower compile. The default setting is on.

#pragma optimize(peep_hole => on| off)   cpm

When on, the compiler performs a variety of machine-code translations, generating smaller and faster code. The default setting is on.

#pragma optimize(jump => on | off)   cpm

When on, the compiler will rearrange loops to eliminate as many jumps as possible, thus generating faster code. The default setting is on.

#pragma optimize(loop => on | off)   cpm

When on, the compiler uses the loop depth when eliminating common sub-expressions and performing jump optimizations. The result of this optimization is faster, but potentially larger, code. The default is on.

#pragma optimize(alias => on | off)   cpm

When on, this allows the compiler to assume that variables in a procedure will not also be used indirectly with a pointer in the same procedure. This assumption is not strictly allowed in ANSI C but is correct for all meaningful programs. The default setting is on.

#pragma optimize(cpu => 86 | 286 | 386 | 486) cpmw

This controls the instructions used by the code generator, by declaring the processor to be used. The default is cpu=>286. Ignored in this release.

#pragma optimize(copro => 87 | 287 | 387)   cpmw

This controls the instructions used by the code generator, by declaring the math coprocessor to be used. The default is cpu=>287. Ignored in this release.

Option #pragmas

#pragmas with the class name *option* control language-dependent options, such as SoftVelocity extensions. The following option #pragmas are available:

```
#pragma option(ansi => on | off)
#pragma option(bit_opr => on | off)
#pragma option(incl_cmt => on | off)
#pragma option(lang_ext => on | off)
#pragma option(min_line => on | off)
#pragma option(nest_cmt => on | off)
#pragma option(pre_proc => on | off)
#pragma option(uns_char => on | off)
```

Note:

A pragma can be used in the Project language, C++ code, Modula-2 code, or Clarion code. Some only work in certain places. A 'P' to the right of the pragma indicates it can be used in the Project language, a 'C' indicates it can be used in C++ code, a 'M' indicates it can be used in Modula-2 code, and a 'W' indicates that it can be used in Clarion code (CLW files).

#pragma option(ansi => on | off) cp

This #pragma is available under C and C++ Only. If it is set to on, ANSI keywords only are allowed. The default setting is off.

#pragma option(lang_ext => on | off) cp

This #pragma is available under C and C++ Only. The following constructs are not valid under ANSI C, but are included in SoftVelocity C and C++ when this #pragma is not set on:

· A type cast yields an lvalue if the operand is an lvalue.

· Procedures can be initialized with binary machine code.

· The relational operators (>,>=,<=,<) allow the operators to be a mixture of integer and pointer operands.

· Bitfields in C can have type char and unsigned char.

· Relative pointers.

The default setting is on.

#pragma option(nest_cmt => on | off) cp

This #pragma is available under C and C++ Only. When on, you can nest comments without causing an error message. For example:

```
/* This is a test comment
/* This is a nested comment */
*/
```

When off, nested comments cause an error message. The default is off, allowing the compiler to trap unterminated comments more easily and make it conform with ANSI C.

#pragma option(uns_char => on | off) cp

This #pragma is available under C and C++ Only. When on, types declared as char lie between 0 and 255. When off, values declared as char lie between -127 and 128. The default setting is off.

#pragma option(pre_proc => on | off) cp

This #pragma is available under C and C++ Only. When on, the compiler produces preprocessor output in a file with the same name but with extension .i. This output file makes it easy to debug the result of macro expansions. The default setting is off.

#pragma option(incl_cmt => on | off) cp

This #pragma is available under C and C++ Only. When on, comments are preserved in preprocessor output. The default setting is off.

This #pragma has no effect unless #pragma pre_proc is on.

#pragma option(min_line => on | off) cp

This #pragma is available under C and C++ Only. When on, the preprocessor minimizes the number of blank lines in output. The default setting is on.

This #pragma has no effect unless #pragma pre_proc is on.

#pragma option(bit_opr => on | off) pm

This #pragma is available under Modula-2 only. It allows bitwise operations on cardinals:

```
( a AND/OR b, NOT a ).
```

The default setting is off.

Project #pragmas

A #pragma with the class name *project* is used to pass information from a compile to the Project System. The value of the #pragma should be a string, which is then stored in the object file for use by the Project System. Whenever an object file is added to the link list, the text specified using this #pragma is executed as a Project System command. For example, if a header file includes the line:

```
#pragma project("#set myflag=on")
```

then whenever a source file that includes this header file is included in a project, the Project System macro myflag will be set. This might be used for processing later in the project file.

This #pragma may only appear in source files, not in a project file.

Save/Restore #pragmas

The save #pragma saves the entire #pragma state, so you can later restore it with a restore #pragma. The save and restore #pragmas work in a stack-like manner, thus allowing you to nest them. For example:

```
/*save the #pragma state and enable the interruptconvention*/

#pragma save

#pragma call(interrupt => on)

/* interrupt functions are specified here */

#pragma restore
```

There is no limit on the number of saves, except the amount of memory available. These #pragmas may be used in source files or in a project file.

Warn #pragmas

#pragmas with the class name *warn* control the generation of compiler warnings. These #pragmas are only available under the project language and C/C++.

The warnings given by SoftVelocity C and C++ help you to check, as far as possible, common coding mistakes. Since no compiler can determine your intentions, you may get warnings even if your code is correct. Your code may generate some warnings more than others, so SoftVelocity allows you to customize which warning checks are performed.

You can set each of the warning options to on, off, or err. When on, SoftVelocity checks the code for that warning and reports the problem, but the problem does not stop the compile or linking. When off, SoftVelocity ignores the warning. When err, SoftVelocity checks the code for the warning, reports the problem, and does not allow linking until you have fixed the problem.

Note: SoftVelocity C and C++ check the code and produce a warning for a good reason. Indeed, to use your non-ANSI C code, SoftVelocity C uses a minimal set of warning messages by default. You should, therefore, think twice before turning off any of the default warning messages. We advise that you keep all the warnings either on or err.

The following warn #pragmas are available:

```
#pragma warn(wacc => on | off | err)

#pragma warn(wait => on | off | err)

#pragma warn(wall => on | off | err)

#pragma warn(watr => on | off | err)

#pragma warn(wcic => on | off | err)

#pragma warn(wcld => on | off | err)

#pragma warn(wclt => on | off | err)

#pragma warn(wcne => on | off | err)

#pragma warn(wcor => on | off | err)

#pragma warn(wcrt => on | off | err)

#pragma warn(wdel => on | off | err)

#pragma warn(wdne => on | off | err)

#pragma warn(wdnu => on | off | err)

#pragma warn(wetb => on | off | err)

#pragma warn(wfnd => on | off | err)

#pragma warn(wftn => on | off | err)

#pragma warn(wnid => on | off | err)

#pragma warn(wnre => on | off | err)

#pragma warn(wnrv => on | off | err)

#pragma warn(wntf => on | off | err)

#pragma warn(wovl => on | off | err)

#pragma warn(wovr => on | off | err)

#pragma warn(wpcv => on | off | err)

#pragma warn(wpic => on | off | err)

#pragma warn(wpin => on | off | err)

#pragma warn(wpnd => on | off | err)

#pragma warn(wpnu => on | off | err)
```

```
#pragma warn(wprg => on | off | err)

#pragma warn(wral => on | off | err)

#pragma warn(wrfp => on | off | err)

#pragma warn(wsto => on | off | err)

#pragma warn(wtxt => on | off | err)

#pragma warn(wubd => on | off | err)

#pragma warn(wvnu => on | off | err)
```

Note:

A pragma can be used in the Project language, C++ code, Modula-2 code, or Clarion code. Some only work in certain places. A 'P' to the right of the pragma indicates it can be used in the Project language, a 'C' indicates it can be used in C++ code, a 'M' indicates it can be used in Modula-2 code, and a 'W' indicates that it can be used in Clarion code (CLW files).

#pragma warn(wall => on | off | err)   cp

This #pragma affects the settings of all the warnings. If set to on or err, all warnings will be enabled.

#pragma warn(wpcv => on | off | err)   cp

Pointer conversion. When on or err, the compiler checks for a conversion between two incompatible pointer types, or between a pointer and an integral type. The default setting is on.

#pragma warn(wdne => on | off | err)   cp

Declaration has no effect. When on or err, the compiler checks for a declaration that has no meaning, for example, long int;. A declaration should contain a variable declarator, a structure or union tag, or members of an enumeration. The default setting is on.

#pragma warn(wsto => on | off | err)   cp

Storage class redeclared. When on or err, the compiler checks that you have not declared the same variable differently within your program. For example:

```
int x; /* External linkage */

static int x; /* Internal linkage */
```

The static storage class always takes preference. The default setting is on.

#pragma warn(wtxt => on | off | err)   cp

Unexpected text in preprocessor command. When on or err, the compiler checks for a new line character terminating a preprocessor command. The default setting is on.

#pragma warn(wprg => on | off | err)   cp

Unknown #pragma. When on or err, the compiler checks for foreign #pragmas or mistakes in SoftVelocity C #pragmas. If you are only creating code using SoftVelocity C or C++ #pragmas, you should switch this warning to either on or err. The default setting is on.

#pragma warn(wfnd => on | off | err)   cp

Function not declared. When on or err, the compiler checks for functions that have been called but not declared. If these functions occur, SoftVelocity C assumes that the function is an extern function returning an int. The default setting is off.

#pragma warn(wpnd => on | off | err)   cp

Function prototype not declared. When on or err, the compiler checks whether a function has a prototype associated with it. Prototypes are important to SoftVelocity, since it can not do much type checking without them. You should, therefore, declare prototypes for all functions. It is best to keep this warning on or err. The default setting is off.

#pragma warn(wnre => on | off | err)   cp

No expression in return statement. When on or err, the compiler checks for a return value in a non-void function. You should keep this warning off if you are compiling some old style C code without prototypes. The default setting is off.

#pragma warn(wnrv => on | off | err)   cp

No return value in function. When on or err, the compiler checks for a return statement in a non-void function. The default setting is off.

#pragma warn(watr => on | off | err)   cp

Different const attributes. When on or err, the compiler checks whether a function that expects a pointer to a variable gets a pointer to a constant. The default setting is on.

#pragma warn(wftn => on | off | err)   cp

Far to near pointer conversion. When on or err, the compiler checks for conversion of a 32-bit far pointer to a 16-bit near pointer. The default setting is on.

#pragma warn(wntf => on | off | err)   cp

Near to far pointer conversion. When on or err, the compiler checks for conversion of a 16-bit near pointer to a 32-bit far pointer. The default setting is on.

#pragma warn(wubd => on | off | err)   cp

Possible use of variable before assignment. When on or err, the compiler checks that you have used a local variable before you have given it a value. SoftVelocity checks this warning with a simple scan through the function, which can cause gotos and the like to generate false warnings.

#pragma warn(wpnu => on | off | err)   cpw

Parameter never used in function. When on or err, the compiler checks for a parameter that the code never uses, so declaration of dummy parameters generates warnings. The default setting is off.

#pragma warn(wdnu => on | off | err) cpw

Variable declared but never used. When on or err, the compiler checks whether a local variable has been declared but never used in the function. The default setting is on.

#pragma warn(wcne => on | off | err) cp

Code has no effect. When on or err, the compiler checks statements and the left operand in a comma expression to see if they have no effect. The default setting is on. For example:

```
x y; /* expression has no effect */

f, x; /* left operand has no effect */
```

#pragma warn(wcld => on | off | err) cp

Conversion may lose significant digits. When on or err, the compiler checks for a conversion from long or unsigned long to int or unsigned int. The default setting is on.

#pragma warn(wait => on | off | err) cp

Assignment in test expression. When on or err, the compiler checks for a possible mistyping of the C equality (==) operator. The equality operator contains two =. For example:

```
     if (x=y) printf("X equals Y"); /* is a mistake */
```

The default setting is on.

#pragma warn(wetb => on | off | err) cp

Value of escape sequence is too large. When on or err, the compiler checks that an escape sequence is in the range 0 to 255. The default setting is on.

#pragma warn(wcor => on | off | err) cp

Value of constant is out of range. When on or err, the compiler checks whether an integer constant is in the range of an unsigned long, or a floating point constant is in the range of a long double. The default setting is on.

#pragma warn(wclt => on | off | err) cp

Constant is long. When on or err, the compiler checks for an integral constant that has type long because of its value but does not have an L suffix. The default setting is off.

#pragma warn(wral => on | off | err) cp

Returns address of local variable. When on or err, checks for a return statement that returns the address of a local variable. This causes a problem because C reclaims the variable storage on completion of the function. The pointer, therefore, points at undefined data. The default setting is on.

#pragma warn(wpin => on | off | err)   cp

Default type promotion on parameter. When on or err, the compiler compares the declaration of a parameter in an old-style function definition with the prototype for incompatible argument promotions. For example:

```
int func(char); /* parameter declared as char */

int func(IntegerByPromotion);
char IntegerByPromotion;
       /* INCOMPATIBLE */
{
 ...
}
```

Note: This is a violation of the ANSI C standard regarding compatible function declarations.The default setting is on.

#pragma warn(wpic => on | off | err)   cp

Parameter list inconsistent with previous call. This warning is issued if a parameter declaration is incompatible with the corresponding parameter in a previous function declaration. The default setting is on.

#pragma warn(wnid => on | off | err)   cp

Address for local variable not in DGROUP. When on or err, the compiler checks that a local variable does not have its address taken in small model, when using #pragma data(ss_in_dgroup => off). The default setting is on.

#pragma warn(wrfp => on | off | err)   cp

Function redeclared with fixed parameters. When on or err, the compiler checks for a prototype with a variable number of arguments, but the corresponding function definition specifies a fixed number of arguments. This will work in SoftVelocity C, but it is a violation of the ANSI C rules for compatible function declarations, and therefore, not portable. The default setting is on.

#pragma warn(wvnu => on | off | err)   cp

Local variable never used. When on or err, the compiler checks whether you declare a local variable and assign it a value but never use it. The default setting is on.

#pragma warn(wovr => on | off | err)   cp

Overflow in constant expression. This warning is issued when a constant integer expression overflows. The default setting is on.

#pragma warn(wacc => on | off | err)   cp

Default access specifier used for base class. This warning is issued if a base class specification does not have an access specifier and the default access is used (i.e. public for a struct and private for a class). The default setting is on.

#pragma warn(wdel => on | off | err)   cp

Expression in delete[] is obsolete. This warning is issued if an expression is specified in the square brackets of a delete expression. The expression is ignored. This is obsolete C usage. The default setting is on.

#pragma warn(wovl => on | off | err)   cp

Keyword ‘overload’ is not required. This warning is issued if the keyword overload is specified in C. The use of this keyword is obsolete C usage. The default setting is on.

#pragma warn(wcic => on | off | err)   cp

Constant in code segment requires initialization. This warning is issued if a constant placed in the code segment requires run-time initialization, as may be the case for an object declared const in C, whose initializer is an expression, when the const_in_code #pragma is set on. This situation will lead to a protection violation in OS/2 and Window 3 protected mode applications, so const_in_code should be set off if this warning is encountered. The default setting is on.

#pragma warn(wcrt => on | off | err)   cp

Class definition as function return type, missing ‘;’ after ‘}’?

This warning is issued if a class is defined in a function return type specification. Such a construct is legal, but unusual, and frequently results from omitting a semicolon between a class definition and the following function declaration. The default setting is on.

Predefined Compiler Flags

Pre-defined Flags

Whenever you #compile a program the project system automatically defines a number of flags to ON or OFF, depending on the target system. You may use these predefined flags to control your make process. Here are the flags that you can use in OMIT and COMPILE statements for conditional compilation:

WIDTH32	On for 32-bit applications(deprecated)
CDD	On for Clarion for DOS
_CW_	On for Clarion, version 1.0
CWVER	Four digit number. The top two digits are the major version of Clarion. The lower two digits are the minor version. For the initial release of Clarion 6.0 this is set to 6000, 6.1, 6100, etc.
CLW15	On for Clarion, version 1.5
CLW20	On for Clarion, version 2.0
CLW21	On for Clarion, version 4
_C5_	On for Clarion, version 5
_C55_	On for Clarion, version 5.5 and later
_C60_	On for Clarion, version 6 and later
_C61_	On for Clarion, version 6.1 and later
_C62_	On for Clarion, version 6.2 and later
_C63_	On for Clarion, version 6.3 and later
_C70_	On for Clarion, version 7.0 and later
DEBUG	On for application debug mode
DLL_MODE	On when compiled to link to the runtime DLLs
LIB_MODE	On when building a LIB
ABCDllMode	Used by the ABC template chain on all CLASS definitions to indicate that the CLASS is declared in an external .DLL. A project DEFINE is used to toggle the DLL mode.

Project System Examples

Following is an example of some project system commands that we use here at SoftVelocity to make our file drivers. This example uses a wide variety of project system statements and shows how the project system can be used to control the accuracy and completeness of even the most complicated projects.

These example statements are divided among four files, showing the project system's ability to support structured programming, modularity, and reusability. The files are ALLDRV.PR, which #calls ORACLE.PR (among others), which #includes SQLFILES.PR, which in turn #includes DRVKIT.PI.

ALLDRV.PR

```
#system win dll
#model clarion


#set drvdebug  = full
#set kitdebug  = full


#set release = off
#set fromclw = on
#set incbuildno = off
#set demo = off


#if "%release"="on" #then
  #pragma define(_RELEASE=>on)
  #set incbuildno = on
  #set kitdebug  = off
  #set drvdebug  = off
#endif


#pragma define(DEMO_VERSION=>%demo)




#set domodels=
 `
 #abort on
 #set dowin32=off #set dolib=off
 #call %%prjfile
 #set dowin32=off #set dolib=on
 #call %%prjfile
 #set dowin32=off #set dolib=off
 #abort off
 #set dowin32=on #set dolib=off
 #call %%prjfile
 #set dowin32=on #set dolib=on
 #call %%prjfile
 #abort on
 `

#if #exists btrieve.pr  #then #set prjfile=btrieve.pr %domodels  #endif
#if #exists odbc.pr     #then #set prjfile=odbc.pr %domodels     #endif
#if #exists cla21.pr    #then #set prjfile=cla21.pr %domodels    #endif
#if #exists tps.pr      #then #set prjfile=tps.pr %domodels      #endif
#if #exists dos.pr      #then #set prjfile=dos.pr %domodels      #endif
#if #exists ascii.pr    #then #set prjfile=ascii.pr %domodels    #endif
#if #exists basic.pr    #then #set prjfile=basic.pr %domodels    #endif

#set domodels=
 `
 #set dowin32=off #set dolib=off
 #call %%prjfile
 #set dowin32=off #set dolib=on
 #call %%prjfile
 #set dowin32=off #set dolib=off
 `



#file redirect ts.red
#if #exists sql400.pr   #then #set prjfile=sql400.pr %domodels   #endif
#if #exists oracle.pr   #then #set prjfile=oracle.pr %domodels   #endif
```

<u>ORACLE.PR:</u>

```
#noedit
-------------------------------------------------------------------
-------------------------------------------------------------------
--  ORACLE.PR  Oracle Driver project file
-------------------------------------------------------------------
-------------------------------------------------------------------
#system win dll                              --target OS is windows, dll executable
#model clarion                               --memory model is clarion

--  Set default macro values. These "switches" will control the make process
#set drv        = ORA
```

```
#set trace      = off
#set heapchk    = off
#set drvdebug   = full
#set sqldebug   = full
#set kitdebug   = off
--#set release  = on

#expand ORACLEIN.CPP                          --set %cpath to path where file is created
                                              --%opath to path where file is opened
                                              --%ext to CPP
                                              --%tail to ORACLEIN
                                              --%cdir to directory where file is created
                                              --%odir to directory where file is opened

#set drvdir = %odir                           --save the %odir value
#set sql_type = O                             --set %sql_type to O

-- Define a conditional compile symbol for subsequent compiles.
-- The symbol is DRVSPEC, and its value is "oraclesp.h"
-- DRVSPEC is available for interrogation by the OMIT and COMPILE
-- statements—see the Language Reference for more information.
#pragma define(DRVSPEC=>'"oraclesp.h"')

--Compile the sql modules with appropriate levels of debug code.
#include SQLFILES.PR                       --Execute statements from SQLFILES.PR here.
                                           --All macros, pragmas, and link list are fully
                                           --available to the #included statements.

#compile ORACLEIN.CPP                      --compile the oracle c++ source.
#compile ORAIMPOR.CLW /define (maincode=>off)--and the clarion source.
                                           --both are added to the link list.
#pragma link(ORAIMPOR.RSC)                 --add ORAIMPOR.RSC to the link list.
#pragma link (ORA7WIN.LIB)                 --add ORA7WIN.LIB to the link list.
#pragma link (%lnkpfx%asc.LIB)             --add C70ASC.LIB to the link list.
                                           --%lnkpfx% resolves to C70,

--  Execute the series of statements assigned to drv_Link at the very
--  end of the DRVKIT.PI file. These statements are designed to link
--  and patch the File Driver.
%drv_Link
```

SQLFILES.PR:

```
-- Release version: Disable all debugging and tracing
#if "%release"="on" #then
 #set drvdebug  = off
 #set kitdebug  = off
 #set sqldebug  = off
 #set trace     = off
 #set heapchk   = off
#endif

-- make sure the sql_type switch is explicitly set (no default)
#if '%sql_type' = '' #then
  #error "sql_type must be set"
#endif

-- Default is compact code. Set the DRIVER_COMPACT symbol based on
-- the value of the %compact macro
#if "%compact"="" #then #set compact=on #endif
#if '%compact'='off' #or '%heapchk'='on' #then
 #pragma define(DRIVER_COMPACT=>off)
#else
 #pragma define(DRIVER_COMPACT=>on)
#endif



#include DRVKIT.PI                           --Execute statements from DRVKIT.PI here.
                                             --All macros, pragmas, and link list are fully
                                             --available to the #included statements.

#pragma save                                 --Save the current #pragma settings so they
                                             --can be restored later with #pragma restore.

-- Debugging: Debugger info and Run-time checks
#if "%sqldebug"="" #then #set sqldebug=off #endif            --default is off
#pragma debug(vid=>%sqldebug)
#if "%sqldebug"="full" #then                                 --for full debugging, enable
 #pragma check(index=>on,range=>on,overflow=>on)             --runtime error checks
 #pragma debug(line_num=>on)                                 --and line numbering
#endif
```

```
#pragma warn(wall=>on)                                          --enable all warning msgs


#set srcfile = SAFESTR.CPP                                      --set %srcfile to SAFESTR.CPP
#set dstfile = %sql_type%SAFE.CPP                               --set %dstfile to OSAFE.CPP
--  Execute the series of statements assigned to makesrc in the middle
--  of the DRVKIT.PI file. These statements are designed to get the C++
--  constructor entry point to have a different name for each driver.
%makesrc


#set srcfile = SQLOPEN.CPP                                       -- ditto
#set dstfile = %sql_type%SQLOPE.CPP
%makesrc


#set srcfile = SQLUPDAT.CPP                                      -- ditto
#set dstfile = %sql_type%SQLUPD.CPP
%makesrc


#set srcfile = SQLRETRI.CPP                                      -- ditto
#set dstfile = %sql_type%SQLRET.CPP
%makesrc


#set srcfile = SQLGLOB.CPP                                       -- ditto
#set dstfile = %sql_type%SQLGLO.CPP
%makesrc


#if %system=win #then                                            -- conditionally
 #set srcfile = SQLVIEW.CPP                                      -- ditto
 #set dstfile = %sql_type%SQLVIEW.CPP
 %makesrc
#endif


#pragma restore                                    --restore the #pragma settings saved earlier
#pragma warn(wall=>on)                             --enable all warning messages
```

DRVKIT.PI:

```
#noedit
----------------------------------------------------------------------
----------------------------------------------------------------------
--  DRVKIT.PI   Driver Kit project include file
----------------------------------------------------------------------
----------------------------------------------------------------------


-- DrvKit.Pi contains project statements common to all Driver Kit based
-- File Drivers.  The following settings must be set before including
-- drvkit.pi:
--
--      #set drv        = A 3 or 4 letter driver id (e.g C21)
--      #pragma define(DRVSPEC='"c21specs.h")   -- or appropriate file
--
--      Any other defines that affect the Driver Specification
--
--  Optional:
--
--      #set trace      = on        -- Enable tracing
--      #set heapchk    = on        -- Enable Heap Checker
--      #set common     = on        -- Merge common code
--


-- Release version: Disable all debugging and tracing
#if "%release"="on" #then
 #set drvdebug  = off
 #set kitdebug  = off
 #set trace     = off
 #set heapchk   = off
#endif


-- Windows: Ensure Clarion 4 Windows conventions are adopted
#pragma warn(wall=>on)              --enable all warning messages
#pragma define(__CLARION__=>on)     --set compiler directive switch


#if #not %system=dos #then          --  dos,
 #if "%dowin32"=on #then
  #system win32 dll                 --  win32, or
 #else
  #system win dll                    --  win16
 #endif
 #model clarion

 #if %filetype=dll #then
  #pragma define(_WINDLL=>on)        --set compiler directive switch
 #endif

-- Build the appropriate file names for this driver set.
 #if "%dolib"="" #then
  #set dolib=off                     --default dolib = off
 #endif
 #if "%dowin32"="" #then
  #set dowin32=off                   --default dowin32 = off
 #endif
 #if %dolib=on #then                 --set link (.lib) prefix
  #set lnkpfx = CL
 #else
  #set lnkpfx = CW
 #endif
 #if "%pfx"="" #then
  #set pfx      = %drv%              --set prefix to driver name
 #endif
#else
 #set dolib=off
 #if %model=extendll #then
  #pragma define(_XTDDLL=>on)
 #endif

 #set pfx       = %M%%drv%
 #if '%RWMODE%' = 'on' #then
  #set lnkpfx=drw
 #else
  #set lnkpfx=%clapfx%
 #endif
 #set S = ""                         --set suffix to null
#endif



#set drvname    = %lnkpfx%%drv%%S%       --Put the name together
```

```
#message "Making %drvname% File Driver" --Display status message
#if #not "%inbrowser" #then
  #if #exists %drvname%.ver #then       --Conditionally...
    #compile %drvname%.ver              --  compile the driver
  #endif
#endif



-- Heap Checker: Compile and enable Heap Checker (No Debugging)
#if "%heapchk"="on" #or "%heapdbg"="on" #then
 #set heapchk = on
 #pragma define(HEAPCHK=>on)            --Set compiler directive switch.
 #if "%heapdll"="on" #then              --If dll, then
  #pragma link(%clapfx%hchk.lib)        --add heapchk lib to link list.
 #else
  #pragma save                          --Save current pragma settings.
  #if "%heapdbg"="on" #then             --Conditionally...
   #pragma debug(vid=>full)             -- enable debug code.
  #endif
  #compile HEAPCHK.C     #to %pfx%HCHK.OBJ--Compile heap checker.
  #compile STRCHK.C      #to %pfx%SCHK.OBJ--
  #compile NEW.CPP       #to %pfx%NEW.OBJ
  #pragma restore                         --Restore saved pragma settings.
 #endif
#endif



-- Debugging: Debugger info and Run-time checks
#if "%drvdebug"="" #then #set drvdebug=full #endif--default is "full"
#pragma debug(vid=>%drvdebug)
#if "%drvdebug"="full" #then
 #pragma check(index=>on,range=>on,overflow=>on)  --enable runtimes
 #pragma debug(line_num=>on)                      --enable line nos.
#endif



-- Common Code: Some Driver Kit code is merged when linking multiple
-- File Driver Libraries
#if "%common"="on" #then                          --Conditionally...
 #pragma define(COMMON_CODE=>on)                  -- define compiler switch
#endif
```

```
#if '%drvdir' = '' #then
 #error "drvdir must be set to build lib versions of the drivers"
#endif

-- To get the C++ constructor entry point to have a different
-- name for each of the drivers it is necessary to compile
-- different C++ source modules:
#set makesrc = '
    #if (#not #exists %%drvdir%%%%dstfile) #or (%%drvdir%%%%dstfile #older %%srcfile ) #or (%%src
file #older %%drvdir%%%%dstfile ) #then
     #expand %%srcfile
     #run "copy %%opath %%drvdir%%%%dstfile > NUL "
    #endif
    #compile %%dstfile %%defns
'

-- Driver Kit: Compile Driver Kit sources
#pragma save                                   --First, save current pragma settings
#if #not "%kitdebug"="" #then                  --Conditionally...
 #pragma debug(vid=>%kitdebug)                 -- set debug level
#else
 #pragma debug(vid=>off)
#endif

#set srcfile = DRVL1.C                         --Set srcfile name
#set dstfile = %pfx%L1.C                       --set dstfile name w correct prefix
%makesrc                                       --Execute stmts defined above.

#if #not "%nocommon" ="on" #then --if common code
 #set srcfile = DRVSTATE.C                     --make DRVSTATE
 #set dstfile = %pfx%STAT.C
 %makesrc

 #if #not (%system=dos) #then                  --if system is not DOS
  #set srcfile = DRVVIEW.CPP                   --make DRVVIEW
  #set dstfile = %pfx%VIEW.CPP
  %makesrc

  #if #not (%filetype=dll) #then               --Conditionally...
   #pragma define(DRV_HAS_LIBMAIN=>on)         -- set compiler switch
```

```
    #endif


    #set srcfile = DRVW.C
    #set dstfile = %pfx%W%dolib%.C
    #set defns = '/define(_LIB_TARGET=>%dolib%)'
    %makesrc
    #set defns = ''


    #set srcfile = DRVWUTIL.C                    --make DRVWUTIL
    #set dstfile = %pfx%WUTI.C
    %makesrc


    #if #not %dolib% #then                       --Conditionally
     #set srcfile = DRVDIAL.CLW                  -- make DRVDIAL
     #set dstfile = %pfx%DIAL.CLW
     #set defns = '/define(maincode=>off)'
     %makesrc
     #set defns = ''
     #pragma link(%pfx%dial.rsc)                 --Add DIAL to link list
    #endif
   #endif
  #endif
  #if "%trace"="on" #then                        --Conditionally...
   #pragma save, define(TRACE=>on)               -- save settings...
   #set srcfile = DRVTRACE.C                     -- make DRVTRACE
   #set dstfile = %pfx%TRAC.C
   %makesrc
  #endif


  #set srcfile = DRVPIPE.C                       --make DRVPIPE
  #set dstfile = %pfx%P%dolib%.C
  #set defns = '/define(_LIB_TARGET=>%dolib%)'
  %makesrc
  #set defns = ''


  #if "%trace"="on" #then
   #pragma restore                               --restore saved pragma settings
  #endif
  #pragma restore


  -- Build Macro drv_Link to be used later in this process
```

```
-- to link and patch the File Driver:
#set drv_Link =
'
 #pragma link_option(share_const=>on)

 #if #not (%%system=dos) #then
  #if "%%dolib"="on" #then
   #dolink %%drvname%%.lib
  #else
   #implib %%drvname%%.lib %%drvname%%.exp
   #if define(_CW15)=on #then
    #pragma linkfirst(idll%%S%%w.obj)
   #else
    #pragma linkfirst(icwdll.obj)
   #endif
   #pragma link(win%%S%%.lib)
   #pragma link(cwrun%%S%%.lib)
   #pragma link_option(decode=>off)
   #dolink %%drvname%%.dll
  #endif

  #if "%%make" #and #not "%%dolib"="on" #then
   #exemod %%drvname%%.dll %%drvname%%.exp %%drvname%%.map
  #endif
 #else
  #if %%filetype=dll #then
   #implib %%drvname%%.lib %%drvname%%.exp
  #endif
  #set tscla    = on
  #set tscpp    = off

  #link %%drvname%%

  #if "%%make" #and (%%filetype=dll) #then
   #expand %%drvname%%.dll
   #run "mkdriver %%cpath > NUL"
  #endif
 #endif
'
```

Module Definition Files (.EXP Files)

A module definition file describes the name, attributes, exports, and other characteristics of a dynamic-link library for Microsoft Windows. This file is required for Windows.

A module definition file (.EXP) is generated whenever you make a new project, or a project whose target type, operating system, or run-time library has changed.

Module Definition File Syntax

A module definition file contains one or more statements. Each statement defines an attribute of the executable file, such as its module name, the attributes of program segments, and the numbers and names of exported symbols. The statements and the attributes they define are listed below:

Statement	**Attribute**
NAME	Names the application
LIBRARY	Names the dynamic-link library
HEAP_COMMIT	Amount of heap committed
HEAP_RESERVE	Amount of heap reserved
STACK_COMMIT	Amount of stack committed
STACK_RESERVE	Amount of stack reserved
IMAGE_BASE	Module base memory location
DEBUG	Include debug information
LINENUMBERS	Include line number information
SECTION_ALIGNMENT	Multiples of 4096 only
FILE_ALIGNMENT	Multiples of 512 only
EXPORTS	Defines exported functions
IMAGE_VERSION n[.m]	Values of *n* and *m* set the image major and minor version fields in PE optional header respectively. n and m must be decimal format. Default values for these fields are zero (0). *n* and *m* are WORD types and have a maximum value of 65535.

The following rules govern the use of these statements:

- If you use either a NAME or a LIBRARY statement, it must precede all other statements in the module definition file.
- You can include source-level comments in the module definition file, by beginning a line with a semicolon(;). The utilities ignore each such comment line.
- Module definition keywords (such as NAME, LIBRARY, and EXPORTS) must be entered in uppercase letters.
- The EXPORTS statement must appear last.

Example—Module Definition File

The following example gives module definitions for a dynamic-link library:

```
LIBRARY MyDLL
; Sample export file
EXPORTS
 Func1 @1
 Var1  @2
 Func2 @3
 Func3 @4
 Func4 @5
```

The NAME Statement

The NAME statement identifies the file as an executable application (rather than a DLL) and optionally defines the name and application type.

NAME [appname] [apptype]

appname If appname is given, it becomes the name of the application as it is known by the operating system. If no appname is given, the name of the executable file, with the extension removed, becomes the name of the application.

apptype Used to control the program's behavior under Windows. This information is kept in the executable-file header. The apptype field may have one of the following values:

WINDOWAPI
The application uses the API provided by Windows and must be executed in the Windows environment.

GUI
Same as WINDOWAPI.

CUI
The program uses a character based user interface, like DOS.

If the NAME statement is included in the module-definition file, then the LIBRARY statement cannot appear.
If neither a NAME statement nor a LIBRARY statement appears in a module-definition file, NAME is assumed.
The following example assigns the name wdemo to the application being defined:

```
NAME wdemo WINDOWAPI
```

The LIBRARY Statement

The LIBRARY statement identifies the file as a dynamic-link library. The name of the library, and the type of library module initialization required, may also be specified.

LIBRARY [libraryname][initialization]

libraryname If libraryname is specified, it becomes the name of the library as it is known by the operating system. This name can be any valid file name. If no libraryname is given, the name of the executable file, with the extension removed, becomes the name of the library.

initialization The initialization field is optional and can have one of the two values listed below. If neither is given, then the initialization default is INITINSTANCE.

INITGLOBAL
The library-initialization routine is called only when the library module is initially loaded into memory.

INITINSTANCE
The library-initialization routine is called each time a new process gains access to the library.

If the LIBRARY statement is included in a module definition file, then the NAME statement cannot appear.

The following example assigns the name mydll to the dynamic-link module being defined, and specifies that library initialization is performed each time a new process gains access to myDLL:

```
LIBRARY myDLL INITINSTANCE
```

The HEAP_COMMIT Statement

Specifies the amount of heap committed. This statement is not generated from the Clarion environment, but may be specified by manually editing the .EXP file and running the linker standalone. The syntax for the HEAP_COMMIT statement is as follows:

HEAP_COMMIT number

The STACK_COMMIT Statement

Specifies the amount of stack committed. This statement is not generated from the Clarion environment, but may be specified by manually editing the .EXP file and running the linker standalone. The syntax for the STACK_COMMIT statement is as follows:

STACK_COMMIT number

The HEAP_RESERVE Statement

Specifies the amount of heap reserved. This statement is not generated from the Clarion environment, but may be specified by manually editing the .EXP file and running the linker standalone. The syntax for the HEAP_RESERVE statement is as follows:

HEAP_RESERVE number

The STACK_RESERVE Statement

Specifies the amount of stack reserved. This statement is not generated from the Clarion environment, but may be specified by manually editing the .EXP file and running the linker standalone. The syntax for the STACK_RESERVE statement is as follows:

STACK_RESERVE number

The IMAGE_BASE Statement

Specifies the base memory location of the module. This statement is not generated by default from the Clarion environment, but can be implemented using the Rebase template. If no IMAGE_BASE is specified in the EXP then the module is assigned the default address (normally 00400000h) and conflicts are handled automatically by the Windows loader. (Using the Clarion IDE you can add an image_base line to the EXP file in the global embed named "Before the Export List".) The syntax for the IMAGE_BASE statement is as follows:

IMAGE_BASE *address*

where *address* is a 32-bit address specified in decimal or hex. If hex, then the address is followed by an "h". The address must be divisible by 64KB (65,536 or 00010000h). It must be in the range of 00400000h to 70000000h for Windows 9x. Under Windows NT the address lower limit is 00010000h.

For more information search MSDN for "Base Address" or "Rebase".

Example:

IMAGE_BASE 00600000h

Tip

It's best to supply the address in hex since all documentation on the OS will show a hex address and it's easy to tell you've got a good address because it always ends with 4 zeros.

The DEBUG Statement

Specifies the SoftVelocity debug information is included. This statement is not generated from the Clarion environment, but may be specified by manually editing the .EXP file and running the linker standalone. The syntax for the DEBUG statement is as follows:

DEBUG

The LINENUMBERS Statement

Specifies that line number information in CodeView format is included. This statement is not generated from the Clarion environment, but may be specified by manually editing the .EXP file and running the linker standalone. The syntax for the LINENUMBERS statement is as follows:

LINENUMBERS

The SECTION_ALIGNMENT Statement

Specifies the section alignment must be in multiples of 4096. This statement is not generated from the Clarion environment, but may be specified by manually editing the .EXP file and running the linker standalone. The syntax is as follows:

SECTION_ALIGNMENT number

The FILE_ALIGNMENT Statement

Specifies the file alignment must be in multiples of 512. This statement is not generated from the Clarion environment, but may be specified by manually editing the .EXP file and running the linker standalone. The syntax is as follows:

FILE_ALIGNMENT number

The EXPORTS Statement

The EXPORTS statement defines the names and attributes of the functions exported to other modules, and of the functions that run with I/O privilege. The term "export" refers to the process of making a function available to other run-time modules. By default, functions are hidden from other modules at run time.

EXPORTS
 exportdefinitions

The EXPORTS keyword marks the beginning of the export definitions. It may be followed by up to 3072 export definitions, each on a separate line. You should give an export definition for each dynamic-link routine that you want to make available to other modules. The syntax for an export definition is as follows:

entryname [pwords] @number | ? [NODATA]

entryname Defines the function name as it is known to other modules.

pwords Specifies the total size of the function's parameters, as measured in words (the total number of bytes divided by two). This field is required only if the function executes with I/O privilege. When a function with I/O privilege is called, OS/2 consults the pwords field to determine how many words to copy from the caller's stack to the I/O-privileged function's stack.

@number | ? Defines the function's ordinal position within the module-definition table. The @ may be followed by the position number of the function, or it may be followed by a question mark (?) if the position is unknown. The numbers must be in sequence.

NODATA Provided for use by real-mode Windows (optional).

The EXPORTS statement is meaningful for functions within dynamic link libraries, functions which execute with I/O privilege, and call back functions in Windows programs.

For example:

```
EXPORTS
 Func1 @?
 Func2 @?
 CharTest @?
```

The MANIFEST statement

MANIFEST [*file name*]

This directive instructs the linker to add specified manifest *file name* to the executable. If the manifest *file name* is omitted, the linker adds a default manifest. If both the project file and the EXP file contain directives to link the manifest file, the one specified in the project file will be used.

Exporting CLASSes

Exporting CLASS declarations requires a special form of export definition.

You must create two export definitions for the CLASS itself. The first begins with VMT$ followed by the name of the CLASS as the entryname. The second begins with TYPE$ followed by the name of the CLASS as the entryname. These are followed by an export definition for each method in the CLASS to export whose pwords must begin with the name of the CLASS as the first parameter.

For example:

```
EXPORTS
 VMT$MYCLASS @?
 TYPE$MYCLASS @?
 FIRSTMETHOD@F7MYCLASS @?
 SECONDMETHOD@F7MYCLASS @?
```

Special Considerations for One-Piece (Single) Executables

A one-piece executable is defined as a project that has been linked into a single, stand-alone executable. The Clarion runtime library and all of the application's procedure calls and libraries are linked into a single file.

Callback functions are a standard part of Windows programming in most programming languages. A callback function is a PROCEDURE that you (the programmer) write to handle specific situations that the operating system deems the programmer may need to deal with. A callback function is called by the operating system whenever it needs to pass on these situations. Therefore, a callback function does not appear to be part of the logic flow, but instead appears to be separate and "magic" without any logical connection to other procedures in your program.

Callbacks are valid when used in one-piece executables (EXEs), but there is a special case which must be handled in a different manner.

Here is the case: If the EXE makes some call to the Operating System, the Operating System starts a new thread inside this call, and then calls to a passed callback function. Using this program design, the one-piece EXE must be converted to a DLL linked in local mode, and a starter EXE must be created, using an External link to the DLL entry point that is used to load and run the one-piece DLL.

The following approach demonstrates how this is done.

The one-piece EXE must be converted to a DLL linked in local mode.

The Local mode DLL must export the name of the entry point's procedure and the following names from the RTL:

```
__checkversion
__sysstart
__sysinit
_exit
Cla$code
Cla$init
Wsl$Closedown
```

Here is an example of the export file (EntryPoint is the procedure entry into the DLL)

```
--------------------------------------
EXPORTS
EntryPoint@F @?
__checkversion @?
__sysstart @?
__sysinit @?
_exit @?
Cla$code @?
Cla$init @?
Wsl$Closedown @?
```

In this example, the entry point procedure name in the Local DLL is: "EntryPoint"

Tip

Use the *Inside the Export List* Global Embed to add to your export list within the application.

The starter EXE must use External link mode. The source is written so that it just calls the DLL's entry point procedure.

Example starter EXE code:

```
----------------------------

PROGRAM MAP MODULE('') EntryPoint() END END CODE EntryPoint
```

Version Information Resource Files

The Clarion Project System supports the inclusion of Version Information, conforming to the industry standard script format.

A version script file is simply a text file with the extension of *.Version*. When included into a Clarion project (application or hand coded), the version file stamps, or writes, a variety of information into the target executable. This information can be viewed by right-clicking on the executable file, and selecting Properties from the popup menu. A Version tab should be available with the designated version information.

More detail regarding the standard format of the version info script can be found at the following URL:

Microsoft Version Control Information

Clarion also adds the following exceptions to this standard:

1. A LANGUAGE directive can precede the Version script as follows:

```
LANGUAGE <language code>

VS_VERSION_INFO VERSIONINFO
...
END
```

If the LANGUAGE directive is present in the version file, the language code for the resource target executable is set. This allows a developer to have multiple version info resources for different languages.

2. In the version information group, numbers must use one of the following formats:

- decimal numbers (0-9)
- hexadecimal numbers in C/C++ format (Example: `0x3fL)`
- hexadecimal numbers in Modula-2/Clarion format (Example: `040904E4)`
- binary numbers in Modula-2/Clarion format

3. Strings must be of C/C++ format. The **\u** and **\x** escape characters are not supported in strings.

4. **#include** directives are not supported, but all standard mnemonics for the version info related constants are built in to the compiler.

Version script example:

```
LANGUAGE 0x419


1 VERSIONINFO
 FILEVERSION 1,0,0,1
 PRODUCTVERSION 1,0,0,1
 FILEFLAGSMASK 0x3fL
 FILEFLAGS 0
 FILEOS VOS__WINDOWS32
 FILETYPE VFT_APP
 FILESUBTYPE 0x0L
BEGIN
    BLOCK "StringFileInfo"
    BEGIN
        BLOCK "040904E4"
        BEGIN
            VALUE "CompanyName", "\0"
            VALUE "FileDescription", "This just a test\0"
            VALUE "FileVersion", "1, 0, 0, 1\0"
            VALUE "InternalName", "Version Info Script Example\0"
            VALUE "LegalCopyright", "Copyright (C) 2003\0"
            VALUE "LegalTrademarks", "\0"
            VALUE "OriginalFilename", "TEST\0"
            VALUE "ProductName", "Version Info Script compiler\0"
            VALUE "ProductVersion", "1, 0, 0, 1\0"
        END
    END
    BLOCK "VarFileInfo"
    BEGIN
        VALUE "Translation", 0x409, 1252
        VALUE "Translation", 0x419, 1251
        VALUE "ÒÅÑÒ", 0x409, 1111
    END
END
```

Tip

This file is a working example. You can use this as a template for your real world version script files. Simply copy this example to a text file, name it *yourfilename*.version, and include it in the **Library, object, and resource files** section of the Project Tree.

Multi Language Programming

Overview

SoftVelocity has 32-bit C++ and Modula-2 compilers that can integrate into the Clarion environment. The 32-bit compilers generate object code for the Windows 95/NT/2000/XP environments. You must be running your development environment under a 32-bit environment to generate applications for a 32-bit environment.

Clarion also has object code generation capability that rivals that of many C compilers. You can then enhance that application with any low-level functions you need. These 3rd Generation Language (3GL) compilers enable the developer to include 3GL code modules directly into a Clarion project, giving unparalleled functionality and versatility. This mix of a Rapid Application Development (RAD) 4GL (4th Generation Language)—Clarion—and traditional 3GL compilers, makes Clarion an exceptional application development tool.

So why use C, C++, Pascal, or Modula-2 code at all in a Clarion application? Because there are libraries available (statistical, financial, graphics, communications, and many more) which could significantly cut the development time of an application that requires these capabilities. Many of these libraries are written in C, and many powerful C++, Pascal, and Modula-2, libraries are also available. Clarion allows you to use these libraries in their "native" form without "re-inventing the wheel."

Throughout this topic, we assume you have a good knowledge of the Clarion development environment, the Clarion language, and the 3GL in question. The code examples assume that you use a SoftVelocity compiler (other compiler’s requirements are also discussed). Since Clarion uses SoftVelocity code generation and linking technology, it is easiest to link code produced with SoftVelocity Compilers to Clarion applications. Clarion can link code produced by other third party compilers; however, some care is required as well as a good understanding of the operation of both compilers. It is not generally possible to directly link C++ code produced by C++ compilers other than SoftVelocity.

Compiler Integration

With the SoftVelocity 3GL compilers installed, the Clarion development environment takes all the action necessary to call the correct compiler for each source module in the application. You cannot mix languages in a single source module; however, an application can contain any number of source modules written in any of the 3GLs or Clarion.

The development environment calls the correct compiler for each module at compile time by looking at the source file extensions, as follows:

Source File Extension	Compiler Called
.CLW	Clarion
.CPP or .C	C++
.MOD	Modula-2

Source files with any other extensions will generate an ‘Unknown compiler for ...’ error message at compile time.

The Development Environment will also ensure that all modules are linked correctly and that the SoftVelocity SmartLinker is given all the information that it requires.

Integrating 3GL Modules into Clarion Projects

Using the Application Generator:

1. Create an "Include file" containing the function prototypes for the Clarion compiler.

   You MUST prototype your functions if you intend to call them from Clarion code (see Procedure Prototyping in the Language Reference). The include file should contain prototypes for all 3GL functions called from Clarion code. Each 3GL module should have its own include file.

   The include file is put in the Generated source without modification by step 3 of this process. The generated Clarion code for your include file appears in the Global Map something like this:

```
MODULE(‘module name’)
INCLUDE(‘YourInc.Inc’)
END
```

   The Application Generator generates the MODULE and END statements. Failure to correctly prototype your functions will almost certainly result in a General Protection Fault at run time.

2. Use the Text Editor to write your 3GL code.

   Be sure to save the code with a file extension that the compiler can recognize (.C, .CPP, .MOD, or .PAS).

3. Add the module to the application as an **External Source Module**.

   Select **Application > Insert Module** from the main menu. Then select class **External Source** from the *Select Module Type* dialog. Press the ellipsis button, and in the Open File dialog, enter the name of the 3GL module in the **Name** field and click on the Open button, Back in the Module Properties dialog, enter the name of the include file in the **Map Include File** field of the *Module Properties* dialog. Press OK to add to the application tree.

4. Compile and run the application.

In a hand-coded Clarion application:

1. Create the function prototypes for the Clarion compiler.

   You must prototype the functions you intend to call from Clarion code (see Procedure Prototyping in the Language Reference). The global MAP structure should contain the prototypes.

   Each 3GL module should have its prototypes in a separate MODULE structure, something like this:

```
MODULE('module name')
MyFunc(*CSTRING),CSTRING,RAW,PASCAL,NAME('_MyFunc')
END
```

   You must include a complete MODULE structure in your Clarion MAP for all your 3GL modules. Failure to correctly prototype your functions will almost certainly result in a General Protection Fault at run time.

2. Use the Text Editor to write your 3GL code, saving the code with a file extension that the compiler can recognize (.C, .CPP, .MOD, or .PAS).
3. Add the module to the Project as an **External Source File**.

   Select **Project > Edit** from the main menu. Highlight **External Source Files** then press the **Add File...** button. Select the 3GL source mofule from the standard file open dialog that appears.

4. Compile and run the application.

Resolving Data Types

The Clarion language defines the data types BYTE, SHORT, USHORT, LONG, ULONG, SREAL, REAL, and STRING which map fairly easily to C, C++, Pascal, and Modula-2 equivalents. Clarion also defines DATE and TIME data types, and GROUP structures, which may be mapped to structures in each language. CSTRING and PSTRING data types are specifically provided by Clarion to simplify interfacing with external functions using C or Pascal conventions.

The DECIMAL, PDECIMAL, BFLOAT4, and BFLOAT8 types are not discussed because it is very unlikely that these types of variables will ever be used in C, C++, Pascal, or Modula-2 code. If data of any of these types does need to be passed to C, C++, Pascal, or Modula-2 code, simply assign the value to a REAL or SREAL variable and pass that to the function (data type conversion is automatically handled in Clarion by the assignment statement).

The table below gives a brief cross reference of the parameters types supported by the Clarion, C++ and Modula-2 compilers; as detailed, some parameters require additional pragma statements to work correctly. The Clarion SIGNED and UNSIGNED data types are equates that change type from LONG and ULONG.

Clarion	C++	Modula-2
BYTE	unsigned char	BOOLEAN
BYTE	unsigned char	SHORTCARD
*BYTE	unsigned char *	var SHORTCARD
USHORT	unsigned short	CARDINAL
*USHORT	unsigned short *	var CARDINAL
SHORT	short	INTEGER
*SHORT	short *	var INTEGER
LONG	long	LONGINT
*LONG	long *	var LONGINT
ULONG	unsigned long	LONGCARD
*ULONG	unsigned long *	var LONGCARD
SREAL	float	REAL
*SREAL	float *	var REAL
REAL	double	LONGREAL
*REAL	double *	var LONGREAL
STRING	*can't pass by value*	*can't pass by value*
*STRING	unsigned int, char *	CARDINAL, ARRAY OF CHAR
*STRING(with RAW)	char[]	var ARRAY OF CHAR
*CSTRING(with RAW)	char[] or char *	var ARRAY OF CHAR
*PSTRING	char[] or Char *	ARRAY OF CHAR
GROUP	struct	var record type
*GROUP(with RAW)	struct *	var record type
*?	void far*	FarADDRESS

UNSIGNED	**unsigned int**
SIGNED	**int**

Clarion STRING variables are normally passed as two parameters: first, a UNSIGNED which contains the length of the data buffer; second, the address of the data. CSTRINGs and PSTRINGs are passed the same as STRINGs (as two parameters). The RAW attribute can be used in the Clarion prototype to pass only the address of the string data to external 3GL functions (Clarion language procedures do not need, or support, RAW).

C and C++ Data Type Equivalents

The following data type equivalents can be used with C or C++ code. These typedefs should appear in the .H header file referenced by the C or C++ code. The CLA prefix is used to avoid name clashes with third party libraries.

```
Typedef  unsigned char   CLABYTE;
Typedef  short           CLASHORT;
typedef  unsigned short  CLAUSHORT;
typedef  long            CLALONG;
typedef  unsigned long   CLAULONG;
typedef  float           CLASREAL;
typedef  double          CLAREAL;
```

Clarion DATE and TIME data types may be passed to C functions as a CLALONG, the CLADATE and CLATIME unions can then be used to resolve the elements of the date or time from the CLALONG value.

```
typedef union {
    CLALONG         n;
    struct {
        CLABYTE     ucDay;
        CLABYTE     ucMonth;
        CLAUSHORT   usYear;
    }               s;
}   CLADATE;
typedef union {
    CLALONG         n;
    struct {
        CLABYTE     ucHund;
        CLABYTE     ucSecond;
        CLABYTE     ucMinute;
        CLABYTE     ucHour;
    }               s;
}   CLATIME;
```

Because of Clarion’s two-parameter method of passing STRINGs, the CLASTRING structure is useful for certain internal uses, but cannot be used to accept parameters from Clarion:

```
    typedef struct {
      char *pucString;
      CLALONG usLen
    } CLASTRING;
```

Clarion STRING variables are not NULL terminated, they are padded with spaces up to the length of the data buffer. The trailing spaces can be removed by using the Clarion CLIP procedure. The following code declares a STRING of 20 characters, assigns some data into it, and passes it as a parameter to a C or C++ function.

```
    StringVar     STRING(20)
      CODE
      StringVar = 'Hello World...'
      C_Write_Function(StringVar)
```

The C or C++ function might be defined as:

```
    extern void C_Write_Function(CLAUSHORT usLen, char *bData)
    {   CLAUSHORT usNdx = 0;

        while (usNdx < usLen)
#ifdef __cplusplus
            cout << bData[usNdx++];
#else
            putchar(bData[usNdx++]);
#endif
    }
```

In the above example, usLen would have a value of 20 and bData would be padded with trailing spaces. This padding would be written to the screen by C_Write_Function(). Many C routines expect a string to be NULL terminated. To address this issue, Clarion provides the CSTRING data type. CSTRING variables are automatically NULL terminated when data is assigned to them. This makes it possible for existing C routines to operate on the data.

A Clarion GROUP may be declared to contain related data. A group is roughly equivalent to a C or C++ struct. When passed as a parameter to a procedure, GROUPs are normally passed as three parameters: first, an UNSIGNED is passed which contains the size of the GROUP; second, the address of the GROUP structure; and third, the address of a buffer containing a type descriptor for the GROUP. The contents of the type descriptor are not discussed here and are subject to change in future versions of Clarion. GROUPs may be nested, and other GROUPs may be defined to assume the same structure as a previously declared GROUP. There are several forms of declaration for Clarion GROUPs:

```
    Struct1        GROUP         ! Struct1 is defined as a GROUP
    ul1              ULONG       ! containing two ULONG values
    ul2              ULONG
                   END
```

This form of definition reserves space for Struct1 and is equivalent to the C definition:

```
      struct {
          CLAULONG    ul1;
          CLAULONG    ul2;
      }   Struct1;
```

In the following example, the declaration of Struct2 declares a GROUP similar to that defined by Struct1, however no space is reserved. In practice there need not be any instances of Struct2 defined.

```
  Struct2          GROUP,TYPE        ! Struct2 is declared as a GROUP
  ul3                ULONG           ! containing two ULONG values
  ul4                ULONG
                   END
```

The corresponding C definition is:

```
      typedef struct {
          CLAULONG    ul3;
          CLAULONG    ul4;
      }   Struct2
```

In the following example, the definitions of Struct3 and Struct4 define them to be LIKE(Struct2), i.e. of the same internal structure. In order to distinguish members of Struct3 and Struct4 from those of Struct2 the S3 and S4 prefixes must be used. Struct3 and Struct4 define instances of Struct2 (which is not necessarily defined anywhere). In both cases space is reserved.

```
      Struct3         LIKE(Struct2)
      Struct4         LIKE(Struct2)
```

The corresponding C definitions are:

```
      typedef  Struct2  Struct3;
      typedef  Struct2  Struct4;

      Struct3     S3;
      Struct4     S4;
```

Clarion GROUP declarations may be nested, for example:

```
      Struct5         GROUP,TYPE          ! Struct5 is defined as a GROUP
      Struct6           GROUP            ! containing a nested GROUP
      ul5                 ULONG
      ul6                 ULONG
                        END
                      END
```

The equivalent C declaration is:

```
typedef struct {
    struct {
        CLAULONG    ul5;
        CLAULONG    ul6;
    }   Struct6;
}   Struct5;
```

Modula-2 Data Type Equivalents

The following data type equivalents are used with Modula-2 code. These definitions should appear in the Modula-2 definition module referenced by the Modula-2 code. These should be used to define parameter and return types of procedures that will be called from Clarion code.

```
 CONST
    BYTE    ::= BYTE;
    SHORT   ::= INTEGER (16-bit);
    USHORT  ::= CARDINAL (16-bit);
    LONG    ::= LONGINT;
    ULONG   ::= LONGCARD;
    SREAL   ::= REAL;
    REAL    ::= LONGREAL;
```

Clarion DATE and TIME data types may be passed to Modula-2 procedures as a LONG, the DATE and TIME RECORDs can then be used to resolve the elements of the date or time from the LONG value.

```
    DATE = RECORD
      CASE : BOOLEAN OF
      | TRUE:
        l         : LONG;
      ELSE
        ucDay     : BYTE;
        ucMonth   : BYTE;
        usYear    : SHORT;
      END
    END;
    TIME = RECORD
       CASE : BOOLEAN OF
      | TRUE:
        l         : LONG;
      ELSE
        ucHund    : BYTE;
```

```
              ucSecond  : BYTE;
              ucMinute  : BYTE;
              ucHour    : BYTE;
            END
          END;
```

Clarion STRINGs are passed in the same manner as Modula-2 open ARRAY OF CHAR parameters with the call(o_a_copy=>off) pragma in effect (the length and the address of the string are passed).

The following example code declares a string of 20 characters, assigns some data into it and passes it as a parameter to a Modula-2 procedure

```
      MAP
        MODULE('M2_Code')
          M2_Write_Proc(*STRING), NAME('M2_Code$M2_Write_Proc')
        END
      END
StringVar    STRING(20)
      CODE
      StringVar = 'Hello World...'
      M2_Write_Proc(StringVar)
```

The Modula-2 procedure might be defined as:

```
DEFINITION MODULE M2_Code;
(*# save, call(o_a_copy=>off) *)
PROCEDURE M2_Write_Proc(StringVar: ARRAY OF CHAR);
(*# restore *)
END M2_Code.
```

Note that Clarion STRINGs are not NULL terminated, they are padded with spaces up to the length of the data buffer. In the above example, StringVar would be padded with spaces up to a length of 20 characters. Variables of type CSTRING are automatically NULL terminated when data is assigned to them. This makes it possible for existing Modula-2 routines to operate on the data.

A Clarion GROUP is roughly equivalent to a Modula-2 RECORD. There are several forms of declaration for Clarion GROUPs. The following conforms to the Modula-2 declaration of the DATE type above:

```
DateType GROUP
n          LONG
d          GROUP,OVER(n)
ucDay        BYTE
ucMonth      BYTE
usYear       SHORT
           END
         END
```

The OVER attribute is used to ensure that n and d occupy the same memory, the total size of the group is the size of the member n. When passed as parameters, GROUPs are normally passed as three parameters: first, an UNSIGNED is passed which contains the size of the GROUP; second, the address of the GROUP structure, and third, the address of a buffer containing a type descriptor for the GROUP. The contents of the type descriptor are not discussed here and are subject to change in future versions of Clarion. You may use the RAW attribute in your Clarion prototype for the Modula-2 procedure to instruct the compiler to pass only the address of the GROUP, otherwise you must define your Modula-2 procedure to take 2 extra parameters:

```
MAP
  MODULE('M2_Code')
    M2_Proc1(*GROUP)
    M2_Proc2(*GROUP), RAW
  END
END
```

The corresponding Modula-2 definition module would contain:

```
DEFINITION MODULE M2_Code;
  TYPE
    GROUP = RECORD
      (* Members *)
    END;
  PROCEDURE M2_Proc1(Len: USHORT;  VAR Data: GROUP; TypeDesc: ADDRESS);
  PROCEDURE M2_Proc2(VAR Data: GROUP);
END M2_Code.
```

Pascal Data Type Equivalents

The following data type equivalents can be used with Pascal code. These should be placed in the Pascal interface unit referenced by the Pascal code. These should be used to define parameter and return types of procedures that will be called from Clarion code.

```
ALIAS
  SHORT   = INT16;
  USHORT  = INT16;
  LONG    = INTEGER;
  ULONG   = INTEGER;
  SREAL   = SHORTREAL;
```

Clarion DATE and TIME data types may be passed to Pascal procedures as a LONG, the DATE and TIME records can then be used to resolve the elements of the date or time from the LONG value.

```
DATE = RECORD
  CASE BOOLEAN OF
  TRUE:
    (n        : LONG);
  FALSE:
    (ucDay    : BYTE;
     ucMonth  : BYTE;
     usYear   : SHORT);
  END;

TIME = RECORD
  CASE BOOLEAN OF
  TRUE:
    (n        : LONG);
  FALSE:
    (ucHund   : BYTE;
     ucSecond : BYTE;
     ucMinute : BYTE;
     ucHour   : BYTE);
  END;
```

Because of Clarion's two parameter method of passing STRINGs, the STRING structure is useful for certain internal uses, but cannot be used to accept parameters from Clarion:

```
TYPE
  STRING = RECORD
    usLen     : USHORT;
    pucString : ^CHAR;
  END;
```

Clarion PSTRINGs are passed by address in the same manner as Pascal STRING parameters with the call(s_copy=>off) pragma in effect (the length and the address of the string are passed).

The following example code declares a string of 20 characters, assigns some data into it, and passes it as a parameter to a Pascal procedure:

```
    MAP
      MODULE('Pas_Code')
        Pas_Write_Proc(*PSTRING), NAME('Pas_Code$Pas_Write_Proc')
      END
    END
StringVar    PSTRING(20)
    CODE
    StringVar = 'Hello World...'
    Pas_Write_Proc(StringVar)
```

The Pascal procedure might be defined as:

```
INTERFACE UNIT Pas_Code;
(*# save, call(s_copy=>off) *)
PROCEDURE Pas_Write_Proc(StringVar: STRING[HIGH]);
(*# restore *)
END.
```

A Clarion GROUP is roughly equivalent to a Pascal RECORD. There are several forms of declaration for Clarion GROUPs. The following duplicates the Pascal declaration of the DATE type above:

```
DateType GROUP
n          LONG
d          GROUP,OVER(n)
ucDay        BYTE
ucMonth      BYTE
usYear       SHORT
           END
         END
```

The OVER attribute is used to ensure that n and d occupy the same memory, the total size of the group is the size of the member n. When passed as parameters, GROUPs are normally passed as three parameters: first, a USHORT is passed which contains the size of the GROUP; second, the address of the GROUP structure; and third, the address of a buffer containing a type descriptor for the GROUP. The contents of the type descriptor are not discussed here and are subject to change in future versions of Clarion. You may use the RAW attribute in your Clarion prototype for the Pascal procedure to instruct the compiler to pass only the address of the GROUP, otherwise you must define your Pascal procedure to take 2 extra parameters:

```
MAP
  MODULE('Pas_Code')
    Pas_Proc1(*GROUP)
    Pas_Proc2(*GROUP), RAW
  END
END
```

The corresponding Pascal interface unit might be:

```
INTERFACE UNIT Pas_Code;
  TYPE
    GROUP = RECORD
      (* Members *)
    END;
  PROCEDURE Pas_Proc1(Len: USHORT;  VAR Data: GROUP; VAR TypeDesc);
  PROCEDURE Pas_Proc2(VAR Data: GROUP);
END.
```

Prototyping 3GL Functions in Clarion

- The only thing necessary to be able to use any of the C standard library functions in Clarion code is the addition of the function’s Clarion language prototype to the Clarion application’s MAP structure. The Clarion prototype tells the compiler and linker what types of parameters are passed and what return data type (if any) to expect from the C function. The PROCEDURE Prototypes section in Chapter 2 of Clarion’s Language Reference discusses the syntax and attributes required to create a prototype of a Clarion procedure. This same syntax is used to create Clarion prototypes of C functions.
- There are four major issues involved in creating a prototype for a C function: calling convention, naming convention, parameter passing, and return data types from functions.
- The calling convention for all the SoftVelocity C standard library functions is the same register-based calling convention used by Clarion. Therefore, there is no need to use the C or PASCAL attributes in any standard C library function’s Clarion prototype.
- The SoftVelocity C compiler’s naming convention is the normal C convention. This means an underscore is automatically prepended to the function name when compiled. The Clarion NAME attribute is usually used in the prototype to give the linker the correct reference to a C function without requiring the Clarion code to use the prepended underscore. For example, the C function "access" is actually named "_access" by the compiler. Therefore, the NAME(‘_access’) attribute is required in the prototype (unless you want to refer to the function in Clarion code as "_access").
- Each parameter passed to a C function must appear in its Clarion prototype as the data type of the passed parameter. Parameters are passed in Clarion either "by value" or "by address."
- When a parameter is passed "by value," a copy of the data is received by the function. The passed parameter is represented in the prototype as the data type of the parameter. When passed "by address," the memory address of the data is received by the function. The parameter is represented in the prototype as the data type of the parameter with a prepended asterisk (*). This corresponds to passing the C function the pointer to the data.

Parameter Data Types

Parameter data type translation is the "key" to prototyping C functions in Clarion. The following is a table of C data types and the Clarion data type which should be used in the prototype:

C Data Type	**Clarion Data Type**
Char	BYTE (gets linker warnings - ignore them)
unsigned char	BYTE
Int	SHORT
unsigned int	USHORT
short	SHORT
unsigned short	USHORT
long	LONG
unsigned long	ULONG
float	SREAL
double	REAL
unsigned char *	*BYTE
int *	*SHORT
unsigned int *	*USHORT
short *	*SHORT
unsigned short *	*USHORT
long *	*LONG
unsigned long *	*ULONG
Float *	*SREAL
double *	*REAL
Char *	*CSTRING w/ RAW attribute
struct *	*GROUP w/ RAW attribute

- Since the Clarion language does not have a signed BYTE data type, linker warnings (‘type inconsistency’) will result when you prototype a function which receives a char parameter. As long as you are aware that the C function is expecting a signed value, and correctly adjust the BYTE field’s bitmap to pass a value in the range -128 to 127, this warning may be safely ignored.
- The RAW attribute must be used when a C function expects to receive the address of a CSTRING or GROUP parameter. By default, Clarion STRING, CSTRING, PSTRING, and GROUP parameters are passed (internally) to other Clarion procedures as both the address and length of the string. C functions do not usually want or need the length, and expect to receive only the address of the data. Therefore, the RAW attribute overrides this default.
- If the C function returns void, there is no data returned and the function fits the definition of a Clarion PROCEDURE. If the C function does return data, it is prototyped with the actual data type returned and the function fits the definition of a Clarion PROCEDURE that returns a value and may be called as part of a condition, assignment, or parameter list.

Return Data Types

Return data types from C functions are almost the same as passed parameters:

C Return Type	Clarion Return Type
char	**BYTE (gets linker warnings - ignore them)**
unsigned char	**BYTE**
int	**SHORT**
unsigned int	**USHORT**
short	**SHORT**
unsigned short	**USHORT**
long	**LONG**
unsigned long	**ULONG**
float	**SREAL**
double	**REAL**
unsigned char *	***BYTE**
int *	***SHORT**
unsigned int *	***USHORT**
short *	***SHORT**
unsigned short *	***USHORT**
long *	***LONG**
unsigned long *	***ULONG**
float *	***SREAL**
double *	***REAL**
char *	**CSTRING (pointer automatically dereferenced)**
struct *	**ULONG (gets linker warnings - ignore them)**

As you can see, the Clarion return type for a char * is CSTRING (not *CSTRING as you might expect). This is because the Clarion compiler automatically dereferences the pointer to the data when the function returns (as it does with all the pointer return types).

Notice that the Clarion return data type for struct * is ULONG. This will generate a "type inconsistency" linker warning. This occurs because the Clarion language does not use pointers, and the ULONG is a four-byte integer which can serve as a replacement for a pointer return type. The warning is not a problem and can be safely ignored. You would probably use memcpy() to get at the returned data.

Passing Parameters

Clarion offers two distinct methods of passing parameters to functions or procedures: "passed by value" and "passed by address."

"Passed by value" means that the calling code passes a copy of the data to the called function or procedure. The called code can then operate on the data without affecting the caller’s copy of the data. These parameters are specified by the parameter’s data type in the prototype.

"Passed by address" means that the calling code passes the address of the data to the called function or procedure. With this method, the called function or procedure can modify the caller’s data. These parameters are specified by prefixing the parameter’s data type with an asterisk (*) in the prototype:

```
MAP
 MODULE('My_C_Lib')
  Var_Parameter(*USHORT)      ! Parameter passed by address
  Val_Parameter(USHORT)       ! Parameter passed by value
 END
END
```

These declarations represent the Clarion interface to the functions contained in the C library My_C_Lib. The following example are the equivalent C declarations:

```
void   Var_Parameter(CLAUSHORT *uspVal);
void   Val_Parameter(CLAUSHORT usVal);
```

Clarion parameters "passed by address" are equivalent to pointers to the relevant C type. Clarion "passed by value" parameters are passed in the same way as C and C++ value parameters.

The corresponding Modula-2 definition module would be:

```
DEFINITION MODULE M2_Code;
 IMPORT Cla;
 PROCEDURE Var_Parameter(VAR us: Cla.USHORT);
 PROCEDURE Val_Parameter(us: Cla.USHORT);
END M2_Code.
```

The corresponding Pascal interface unit would be:

```
INTERFACE UNIT Pas_Code;
 IMPORT Cla;
 PROCEDURE Var_Parameter(VAR us: Cla.USHORT);
 PROCEDURE Val_Parameter(us: Cla.USHORT);
END.
```

You cannot pass a Clarion STRING or GROUP by value. For this reason, you must pass STRINGs or GROUPs by address.

Resolving Calling Conventions

Clarion uses the SoftVelocity object code generator, so it uses the same efficient register-based parameter passing mechanism employed by all SoftVelocity languages. If differing calling conventions are used by code compiled by third-party compilers, the results may be unpredictable. Typically, the application will fail at run-time.

To use code produced by compilers other than SoftVelocity, you must ensure that either:

1) The other compiler generates code using Clarion’s (SoftVelocity’s) parameter passing method, or,

2) That Clarion generates code using the other compiler’s parameter method.

You must also ensure that none of the functions return floating-point data types. There is no standard of compatibility between compilers regarding this issue. For example, Microsoft C returns floating-point values in a global variable while Borland C returns them on the stack (SoftVelocity also returns them on the stack but there is no guarantee of compatibility). Therefore, any functions from non-SoftVelocity compilers which must reference floating point values and modify them should receive them "passed by address" and directly modify the value — do not have the function return the value.

Most other compilers don’t provide Clarion-compatible parameter passing conventions, but do provide standard C and Pascal parameter passing mechanisms (passed on the stack). Clarion has the C and PASCAL procedure prototype attributes to specify stack-based parameter passing.

Most non-SoftVelocity C and C++ compilers use a calling convention where parameters are pushed onto the stack from right to left (as read from the parameter list). The Clarion C attribute specifies this convention. Many C and C++ compilers also offer a Pascal calling convention where parameters are pushed left to right from the parameter list. Most other languages on the PC also use this convention. The Clarion PASCAL attribute generates calls using this convention.

In most cases, the C and PASCAL attributes are used in conjunction with the NAME attribute. This is because many compilers prepend an underscore to function names where the C convention is in use, and uppercase function names where the PASCAL convention is in use (Clarion uppercases procedure names also). For example:

```
MAP
 MODULE('My_C_Lib')
  StdC_Conv(UNSIGNED, ULONG), C, NAME('_StdC_Conv')
  StdPascal_Conv(UNSIGNED, ULONG), PASCAL, NAME('STDPASCAL_CONV')
 END
END
```

When the StdC_Conv procedure is called, the ULONG parameter is pushed on the stack followed by the UNSIGNED parameter. When StdPascal_Conv is called, the UNSIGNED parameter is pushed followed by the ULONG parameter. You should be very careful that calling conventions match, otherwise the program may behave unpredictably. When interfacing with code produced by SoftVelocity compilers, the C and PASCAL calling convention attributes are not necessary because Clarion uses the SoftVelocity register-based calling conventions.

When writing SoftVelocity C functions to be called from a Clarion program, the CLA_CONV macro (discussed above) should be used to select the correct naming conventions. The best way of achieving this is to declare any interface functions in a separate header (.H) file and to apply the conventions to these declarations. C++ functions must be declared using "Pascal" external linkage (also discussed above). Modula-2 and Pascal naming conventions are best handled by using the NAME attribute on the prototype.

Resolving Naming Conventions

When linking code produced from different programming tools, it is essential to ensure that the proper naming conventions are used. If differing naming conventions are used, the linker will not be able to resolve references to a name within code (produced by one compiler) and its definition (within code produced by another compiler). In this case, no .EXE will be generated.

Many C compilers (including SoftVelocity) prepend an underscore to the name of each function or variable name. The Clarion NAME attribute simplifies interfacing with code produced by these compilers by explicitly telling the Clarion compiler the function or procedure name to generate for the linker. This allows you to explicitly code the Clarion prototype to follow the C convention. For example:

```
MAP
 MODULE('My_C_Lib')
  StdStr_Parm(STRING), NAME('_StdStr_Parm')
 END
END
```

When the Clarion compiler encounters the StdStr_Parm() procedure, it generates the name _StdStr_Parm in the object code. Although Clarion names are not case sensitive, the name generated using the NAME attribute will appear exactly as specified.

The following C language macro defines the Clarion naming conventions. This macro can be used when declaring C functions to interface with Clarion in order to force the C compiler to generate names following the Clarion naming convention (no prepended underscore and all upper case).

```
#define CLA_CONV name(prefix=>"", upper_case=>on)
```

C++ compilers encode the return and parameter types of a procedure into the name that appears in the object code in a process known as 'name mangling'. Therefore, C++ compiled functions which may be called from Clarion can be declared within a 'extern "Pascal" {...};' modifier, which is the equivalent to the C language CLA_CONV macro (which does not affect the name mangling employed by the C++ compiler). For example:

```
extern "Pascal" void Clarion_Callable_Proc(void);
```

A more flexible form of the above, allowing for compilation by either a C or C++ compiler, is:

```
#ifdef __cplusplus
extern "Pascal" {              /* Force Clarion conventions in C++ */
#else
#pragma save, CLA_CONV         /* Force Clarion conventions in C   */
#endif
void Clarion_Callable_Proc(void);   /* C or C++ declaration   */
#ifdef __cplusplus
}                              /* Restore C++ conventions     */
#else
#pragma restore                /* Restore C conventions      */
#endif
```

This form of declaration usually appears in a header file to be included by any interface code. It ensures that the correct conventions are used when compiled with a C or C++ compiler and eliminates the need to use the NAME attribute on the Clarion language prototype of the procedure or function.

Clarion is a case-insensitive language and the compiler converts the names of all procedures to upper-case. Modula-2 and Pascal, however, are case sensitive and also prefix the name of all procedure names with the name of the module in the form: MyModule$MyProcedure. The way to resolve these differences is to use Clarion’s NAME attribute to specify the full name of the Modula-2 or Pascal procedure to the Clarion compiler:

```
MAP
 MODULE('M2_Code')
  M2_Proc1(*GROUP), RAW, NAME('M2_Code$M2_Proc2')
 END
 MODULE('Pas_Code')
  Pas_Proc1(*GROUP), RAW, NAME('Pas_Code$Pas_Proc2')
 END
END
```

The corresponding Modula-2 definition module might be:

```
DEFINITION MODULE M2_Code;
 TYPE
  GROUP = RECORD
   (* Members *)
  END;
 PROCEDURE M2_Proc1(VAR Data: GROUP);
END M2_Code.
```

The corresponding Pascal interface unit might be:

```
INTERFACE UNIT Pas_Code;
 TYPE
  GROUP = RECORD
   (* Members *)
  END;
 PROCEDURE Pas_Proc1(VAR Data: GROUP);
END.
```

The naming conventions used by Clarion for data differ from those used for PROCEDURES, and are more complex. Therefore, the NAME() attribute should be used to generate a Modula-2 or Pascal-compatible name for any Clarion data that needs to be accessed between languages. Modula-2 and Pascal data names are case sensitive and prefixed with the name of the module and a ‘@’ in the form: MyModule@MyProc.

The EXTERNAL and DLL Attributes

The EXTERNAL attribute is used to declare Clarion variables and functions that are defined in an external library. The DLL attribute declares that an EXTERNAL variable or functions is defined in a Dynamic Link Library (DLL).

These attributes provide Clarion programs with a means of accessing public data in external libraries. The compiler will not reserve space for any variables declared as EXTERNAL. For example:

```
typedef struct {
   unsigned long   ul1;
   unsigned long   ul2;
}   StructType;
#ifdef __cplusplus
extern "C" {     /* Use C naming conventions, which will require use */
#endif          /* of the NAME attribute in the Clarion prototype   */
StructType  Str1;        /* Define Str1              */
StructType  Str2;        /* Define Str2              */
#ifdef __cplusplus
}                       /* Restore C++ conventions    */
#endif
```

The following Clarion declarations are all that is necessary to make Str1 and Str2 available to Clarion programs.

```
StructType  GROUP,TYPE  ! Declare a user defined type
ul1          ULONG
ul2          ULONG
           END
! Declare Str1 and Str2 which are defined in the C module
Str1       LIKE(StructType),NAME('_Str1'),EXTERNAL
Str2       LIKE(StructType),NAME('_Str2'),EXTERNAL
```

The NAME attribute is used to allow the linker to use the C naming convention when referencing Str1 or Str2.

Programming Considerations

Using C++ Class Libraries

There are some limitations that apply to accessing C++ code and data from Clarion. C++ is an object oriented language and includes language features to support classes and objects, polymorphism, operator and function overloading, and class inheritance. None of these features are supported in Clarion as they are in C++. This does not prevent you from taking advantage of these features in a mixed Clarion and C++ application, but it does dictate the nature of the interface code.

Clarion cannot directly access C++ classes, or objects of a class type. Therefore, Clarion programs do not have direct access to the data or functions contained within those classes. To access them, it is necessary to provide a "C-like" interface to the C++ functionality. A C style function can be called from Clarion, which would then be able to access the C++ classes and objects defined within the code, including their public data and methods.

The following example code fragment demonstrates how to code a C++ function that calls a C++ class library. The MakeFileList function may be called directly from Clarion — the DirList constructors and the ReOrder class member may not. The DirList class implements a directory list whose entries may be ordered by name, size or date. The class definition and Clarion callable entry point declarations are as follows (note the use of the 'extern "C"' linkage specifier to force C naming conventions for the Clarion callable functions):

```
//*** DirList Class Definition
class DirList: public List {
public:
    DirList(char *Path, CLAUSHORT Attr, CLAUSHORT Order);
    DirList();
    void ReOrder(int Order);
};

//*** Clarion Entrypoint Declarations
extern "C" {
void MakeFileList(char *Path, CLAUSHORT Attr, CLAUSHORT Order);
}
```

The following code does nothing more than provide entry points for the Clarion code to access the functionality of the DIRLIST class library. Since Clarion performs no name-mangling and cannot access classes or their members, this API is necessarily fairly simple.

```
DirList *FileList = NULL;             // The directory list object

void MakeFileList(char *Path, CLAUSHORT Attr, CLAUSHORT Order)
{   if (FileList != NULL)             // If we have a list
  {   delete FileList;                // invoke class destructor
      FileList = NULL;                // so we can start again
  }
  FileList = new DirList(Path, Attr, Order);
}
```

The following is the corresponding MAP structure prototype to allow Clarion to call the MakeFileList interface function:

```
MAP
  MODULE('DirList')
    MakeFileList(*CSTRING,USHORT,USHORT),RAW,NAME('_MakeFileList')
  END
END
```

One disadvantage of this is that, given a large class library, it appears to involve a lot of extra work to create a suitable interface. In practice, however, it should only be necessary to provide a very small interface to begin taking advantage of an existing C++ class library.

It is not possible to call C++ code compiled using non-SoftVelocity C++ compilers from a Clarion application. C++ modules usually require special initialization — constructors for all static objects must be invoked in the correct order. This initialization process must be performed by the Clarion start-up code. Clarion's startup code automatically performs the necessary initialization for any SoftVelocity C++ modules that are present, but it will not initialize modules compiled with other C++ compilers. Even if the modules did not require initialization, other C++ compilers use different calling and naming conventions, and adopt different internal class structures. This makes it impossible to use C++ class libraries in Clarion applications compiled with a compiler other than SoftVelocity C++.

Summary:

The Clarion API provides a number of features to assist developers who need to interface to code written in other programming languages. With a little care, it is possible to create Clarion interfaces to some extremely powerful external libraries.

When preparing interfaces to libraries written in other languages you should consider the following suggestions:

* Don't write C, C++, Pascal, or Modula-2 functions to return CSTRING variables to Clarion. Have the other language routine place the CSTRING value in a public variable, or pass a *CSTRING (by address) parameter to the C routine to receive the value.
* Don't call Clarion procedures that return STRING variables from other language functions. Have the Clarion procedure place the return value in a public variable or pass a *CSTRING (by address) parameter to the other language procedure.
* For simplicity and efficiency, STRING and GROUP parameters should usually be passed by address with the RAW attribute to ensure only the address is passed.
* Test the application in XLARGE memory model first.

C and C++ Considerations

* If a C or C++ function takes a pointer parameter, the corresponding parameter in the Clarion prototype for that function should be declared as "passed by address" by prefixing the data type with an asterisk (*).
* If a C or C++ function takes a pointer to a GROUP, STRING, PSTRING or CSTRING, you should use the RAW attribute in the Clarion prototype.
* If a C or C++ function takes an ASCIIZ string as a parameter, the corresponding parameter in the Clarion prototype should be *CSTRING.
* If a C or C++ function takes a pointer to a structure as a parameter, the corresponding parameter in the Clarion prototype should be *GROUP.
* Use the header (.H) files as a template for developing a Clarion interface to a C or C++ library that eliminates the need to use the NAME attribute on the Clarion prototype to specify names.
* Use the NAME attribute on the Clarion prototype to specify names for C library functions that do not use the CLA_CONV macro - remember that C names are case sensitive and start with an underscore (_).

Modula-2 and Pascal Considerations

* If a Modula-2 or Pascal procedure takes a VAR parameter, the corresponding parameter in the Clarion prototype for that procedure should be declared as "passed by address" by prefixing the data type with an asterisk (*).
* If a Modula-2 or Pascal procedure takes a VAR parameter for a GROUP, STRING, PSTRING or CSTRING, you should use the RAW attribute in the Clarion prototype.
* If a Modula-2 or Pascal procedure takes a VAR record as a parameter, the corresponding parameter in the Clarion prototype should be *GROUP and the RAW attribute should be used in the prototype.

Additional C++ Considerations

* Use the "Pascal" external linkage specification for your C++ interface functions. This eliminates the need to use the Clarion NAME attribute on the prototype.
* Don’t call C++ class member functions from your Clarion code.
* Don’t try to access C++ objects of class type from your Clarion code.
* Don’t try to access C++ code compiled with a C++ compiler other than SoftVelocity.

Additional Modula-2 Considerations

* Use the definition (.DEF) module as a template for developing a Clarion interface to a Modula-2 library.
* If a Modula-2 procedure takes an ASCIIZ string as a parameter, the corresponding parameter in the Clarion prototype should be *CSTRING.
* Use the NAME attribute to specify names for Modula-2 library procedures -remember that Modula-2 names are prefixed with the module name followed by a ‘$’ and are case-sensitive.

Additional Pascal Considerations

* Use the interface (.ITF) files as a template for developing a Clarion interface to a Pascal library.
* Use the NAME attribute to specify names for Pascal library procedures -remember that Pascal names are prefixed with the module name followed by a ‘$’ and are upper-case.

Thread Model Documentation

The Clarion 7 Threading Model

Clarion Version 6 introduced a more powerful, thread support in the templates and runtime library, and that same standard is applied in Clarion 7.

The latest thread model uses *preemptive t*hreads. Typical Clarion programs won't require more than a "compile and link" to see its benefits.

Some advantages of this model are:

- It is much easier to access COM objects
- You can have threads running independently of other threads.
- Programs are more stable.

This thread model also makes the OLE layer much easier to work with because the object will run on the Clarion thread whereas currently it is run on its own separate thread.

The following links provide in-depth information regarding this issue.

Considerations when creating an OLE object

Data Initialization Schema

Launching a thread - behind the scenes

Multi-Threading Programming

Cooperative Threading in a Preemptive Environment

One of the advantages of Clarion 7 is that it is still possible to create an application that emulates the behavior associated with cooperative threading. Clarion 7 provides the necessary hooks so that you can decide whether your threads are preemptive, cooperative, or a mixture of the two. This topic introduces the concept and provides code samples.

Cooperative Threading Extension templates

All applications created prior to Clarion 7 had all threads other than the first one work in a cooperative manner. All new applications, and applications migrated to Clarion 7 will, by default, generate fully preemptive thread based applications.

If you decide to emulate the cooperative thread system, you will still have to convert some code affected by the change in the underlying thread and memory management model (see the *Multi-Threaded Programming* PDF).

If you use the Global Cooperative Threading extension template you are not limited to only have cooperative threads. You can have some or all threads preemptive. To make a procedure thread preemptive you need to activate the "Preemptive Procedure" procedure extension.

If your application is generated with preemptive threads, you can still make individual threads co-operative by adding the Global Cooperative Threading extension and turning off the cooperative option. Then, on each "Preemptive Procedure" procedure extension, turn off the preemptive feature to make the target procedure thread cooperative.

Data Initialization Schema

With the advent of the new threading model in this release, it is important to understand how threaded and non-threaded data elements are initialized, and in what specific order.

Prior to this release, data elements in respective modules were initialized in the following schema.

- non-threaded data of module1
- non-threaded data of module2
- ...
- non-threaded data of module*n*

- threaded data of module1
- threaded data of module2
- ...
- threaded data of module*n*

With the current release, the data initialization schema is as follows:

- non-threaded data of module1
- threaded data of module1
- non-threaded data of module2
- threaded data of module2
- ...
- non-threaded data of module*n*
- threaded data of module*n*

This means that a problem could arise if you attempt to access a threaded file from the constructor of a non-threaded class module. The initialization sequence for objects has changed so that non-threaded objects within a module are initialized before threaded ones.

There are two solutions to overcome this issue.

1. Make the threaded file *non-threaded*, or:

2. Move the definition for the class into a different module and set its *initialization priority* (init_priority) to a number less than the default initialization of 5.

```
PRAGMA ('define(init_priority=>4)') !add to the top of the module
```

The compiler initialization sequence of data, and calls to constructors is as follows:

1) Patching for threaded data in module-1

2) Patching of non-threaded data in module-1

3) Constructors of non-threaded data in module-1

4) Constructors of threaded data in module-1

.... repeat (1)-(4) for additional modules.

Steps (1) and (2) can be required if FILEs or VIEWs use strings that are imported from external

DLLs.

Launching a thread - behind the scenes

With the advent of two new language statements supporting thread management in Clarion 7 (SUSPEND and RESUME), it is important to understand that there are a few things that are initialized and executed behind the scenes by the runtime library each time a thread is STARTed.

Here is the sequence of actions performed by the launching thread and the runtime library(RTL) each time a thread is STARTed:

1. Launching Thread executes START(ThreadProc)
2. RTL creates the physical thread in suspended state.
3. RTL resumes the launched thread created in step 2.
4. RTL sets an internal semaphore to a nonsignaled state.
5. Launching Thread waits for the semaphore from the RTL.
6. RTL creates instances of threaded variables and calls initialization routines for them.
7. RTL sets the semaphore to signaled state.
8. RTL suspends the launched thread creates in step 2.
9. Launching Thread continues program execution.

The launching thread will continue until it encounters the ACCEPT statement. Upon execution of the ACCEPT statement:

10. RTL resumes the launched thread.
11. RTL calls the entry point of the ThreadProc.

Therefore, a launched thread will remain suspended until the next call to ACCEPT from the launching thread. Only initialization and constructors for threaded variables are executed.

The use of RESUME with the START statement immediately executes Step 10 above without waiting for the call to ACCEPT. In other words, use of RESUME with START does not depend on the ACCEPT statement for resuming thread execution. This allows a new thread to be started from windowless threads.

The same can be said by using the SUSPEND statement immediately after START, e.g., SUSPEND immediately stops thread execution and does not wait for the ACCEPT loop.

C++ and API Considerations

Overview

The Clarion RTL thread handler does not do automatic binding of non-Clarion threads.

This minimizes unnecessary overhead; for example, some COM objects start up to 10 internal threads and Clarion data is not accessed from within them.

If you need access to threaded data or to make calls to Clarion functions from a non-Clarion thread you must use the AttachThreadToClarion() function from the non-Clarion DLL. On the call to AttachThreadToClarion(), instances of threaded variables are allocated and their constructors are called, and so they are available to the thread created by the CreateThread API.

Prototype Information

The AttachThreadToClarion() is prototyped as follows:

Clarion prototype:

```
AttachThreadToClarion (BOOL),PASCAL
```

C/C++ prototype:

```
void PASCAL AttachThreadToClarion (int);
```

Visual Basic prototype:

```
Declare Sub AttachThreadToClarion Lib "C70runx.dll" (ByVal Bool As Boolean)
```

This **AttachThreadToClarion** function must be called by the launching thread to attach it to the thread handler in the runtime library. Without this call, any attempt within the thread to use thread dependent Clarion functions and operations (all string and decimal operators, BIND/EVALUATE, ERROR/ERRORCODE, most FILE statements, etc.) will cause a program termination.

The main thread of Clarion programs and all threads started with the START function do not require this call.

The **AttachThreadToClarion** has a BOOLEAN parameter that is used in the following manner. If **AttachThreadToClarion** is called with a FALSE (0) parameter, the runtime library does not allocate instances of threaded variables for this thread and hence, does not initialize them. If the parameter is TRUE (1), the runtime library allocates instances of threaded variables for the thread and executes the initialization code for them. Any attached threads are detached automatically on their termination.

Preemptive and Cooperative Thread Models

This topic is useful for developers looking to migrate their applications from Clarion 5.5 or earlier to Clarion 7, and who wish to understand the new preemptive threads available under Clarion 7. Be sure to read the "**Multi-Threaded Programming**" PDF for a more detailed look at the changes, and information on how to implement thread synchronization.

Threads and Processes

In simple terms, a process can be defined as an application. More precisely, a process is a single instance of an application. (In some cases, you can launch more than one instance of a given application; each launch spawns a new process.) Each process owns its own register context and virtual address space, which includes code, global data, stack, and heap memory.

A thread is an independent flow of execution within a process. In that sense, you can think of a process as a container for one or more threads. All threads in a process share the same code, global data, stack, and heap memory, but each thread is apportioned its own region in the stack space.

Cooperative Threading

Cooperative threads are the only sorts of threads available in versions of Clarion prior to Clarion 7.

Advantages of Cooperative threads

They are safe and easy to use. You can, generally, access variables and not have to worry about the contents of the variable changing between two lines of code. The following code will never display the message "Boom" under a cooperative threading system.

```
MyVar  LONG  ! No THREAD attribute

LOOP
  MyVar = THREAD()
  IF MyVar ~= THREAD()
    MESSAGE('Boom')
  END
END
```

Disadvantages of Cooperative threads

Only one thread is ever active. In other words, you cannot have a thread that runs independently of all your other threads; for example a thread that is continuously backing up a data file, or running a report in the background.

Accessing COM objects and some Windows API functions require careful coding and the use of LOCKTHREAD and UNLOCKTHREAD.

Preemptive Threads

Preemptive threads run independently of each other. Only one thread at a time has focus, but focus can change from one line of code to the next. The switching between threads is under the control of the OS.

Advantages of Preemptive Threads

Have you ever built an application where users had to wait while the application performed some lengthy calculation or operation? Then you can benefit from a preemptive thread model.

Understanding threads using a modern operating system's multi-threading capabilities properly, is a fundamental step toward creating fast, responsive applications. Clarion 7 makes creating multi-threaded applications easier than ever.

To understand the power of multithreading you need to know something about how the Windows operating system works under the hood. Windows is a preemptive multitasking operating system. The system CPU can do only one thing at a time, but to give the illusion that multiple processes are running simultaneously the operating system splits the CPU time between the various running processes.

The term preemptive means that the operating system determines when each task executes and for how long. Preemptive multitasking prevents one task from taking up all the processor's time.

The operating system allocates small "slices" of CPU time to each of these processes. Because the CPU is very fast, and the time slices can be extremely small, processes appear to run simultaneously. On a multi-processor system things are a bit more complicated but the basic idea is the same—the operating system divides the time of the CPUs among the processes that need it.

Difficulties in Programming Preemptive Threads

First, keeping track of and switching between threads consumes memory resources and CPU time. Each time the CPU switches to another thread, the state of the current thread must be saved (so it can be resumed again later) and the saved state of the new thread must be restored. With too many threads any responsiveness advantages you hoped to gain may be partially nullified by the extra load placed on the system.

Second, programming with multiple threads can be complex. Creating a single extra thread to handle some background calculations is fairly straightforward, but implementing many threads is a demanding task and can be the source of many hard-to-find bugs. A best practices approach to these potential problems is to use multithreading only when it provides a clear advantage, and then to use a few threads as possible.

Third is the question of shared resources. Because they're running in the same process, the threads of a multi-threaded program have access to that process's resources, including global, static, and instance fields. Also, threads may need to share other resources such as communications ports and file handles. You must synchronize the threads in most multi-threaded applications to prevent conflicts when accessing resources, such as deadlocks (when two threads stop as each waits for the other to terminate).

You need to be **very** careful when accessing non-threaded global variables. The following code will generate the message intermittently when using preemptive threads.

```
MyVar  LONG  ! No THREAD attribute


LOOP
  MyVar = THREAD()
  IF MyVar ~= THREAD()
    MESSAGE('Bang')
  END
END
```

You can protect yourself in this situation by using the IMutex interface declared in CWSYNCH.INT to stop other threads executing at the same time (for a short time)

```
MyVar  LONG  ! No THREAD attribute
MyLocker &IMutex
MutexName CSTRING('MyCode')

MyLocker &= NewMutex(MutexName)
LOOP
  MyLocker.Lock()
  MyVar = THREAD()
  IF MyVar ~= THREAD()
    MyLocker.Release()
    MESSAGE('Bang')
  ELSE
    MyLocker.Release()
  END
END
MyLocker.Kill()
```

If you do not protect yourself when accessing global non-threaded variables you will have an application that will appear to work, but will occasionally do something strange that is hard to reproduce with any consistency.

Programming with preemptive threads can be **complex**! Even code as simple as…

```
MyFunc FUNCTION()
MyResponse = RequestCompleted
RETURN

  MyFunc()
  IF MyResponse = RequestCompleted
    !Do Something
  END
```

…may not work how you expected with multiple threads active, if MyResponse does not have the THREAD attribute

Conclusion

Under Clarion 7 you can now take advantage of the power and flexibility of preemptive threads. However, doing so requires that you are very careful with accessing global non-threaded variables.

So what code has to change?

- Code that stores the reference to a threaded variable in a non-threaded variable will not work the same in Clarion 7 as it did in previous versions of Clarion. For an example of this kind of code see ABFILE.CLW in versions of Clarion prior to Clarion 7 where the FileManager (a non-threaded class in standard applications) stores references to its file's record buffer and fields. The file is normally threaded, so the code needed a small adjustment to work in Clarion 7. Hence ABFILE.CLW and the initialization of standard applications have been changed to reflect this fact.
- In versions of Clarion previous to Clarion 7 you did not put the THREAD attribute on a variable that is declared with the EXTERNAL attribute. In Clarion 7 you must put the THREAD attribute on a variable declared with the EXTERNAL attribute when the variable is threaded. Not doing this will cause your application to crash whenever it tries to access the variable from anywhere other than the DLL where the variable is declared.
- In Clarion 5.5 and prior CW versions the runtime library (RTL) had control on thread switching. If the current event was targeted to a WINDOW belonging to a thread other than the previous event was, it generated EVENT:Suspend for previous active thread, EVENT:Resume for the new active thread and then executed the event. This could be any event. In Clarion 7 switching of threads is not within the RTL's control, so EVENT:Resume and EVENT:Suspend events no longer make sense.

Be sure to read the "**Multi-Threaded Programming**" PDF for a more detailed look at the changes, and information on how to implement thread synchronization.

Multi-Threading Programming

Multi-Threading Programming

Contents:

Overview

This topic introduces programming techniques you implement when you program in a preemptive multi-threading environment. It identifies the type of code and objects that require attention, and also provides code examples for adding thread synchronization to properly handle access to shared non-threaded data.

Throughout this topic there are references to static variables. There are three ways of creating a variable in Clarion such that it is static:

- The variable is declared in Global Data and does not have the THREAD attribute;
- The variable is declared in Module Data and does not have the THREAD attribute;
- The variable is declared in Procedure Data and has the STATIC attribute, but does not have the THREAD attribute;

Static variables are potentially dangerous in a preemptive environment, and should be avoided where possible. Later sections in this topic will describe what to do when they cannot be avoided.

What is a threading model?

A threading model describes how different threads work together under a multi-threading operating system. There are two types of threading models: cooperative threading and preemptive threading.

Under the cooperative threading model an application could have multiple threads ‘running’ at the same time. But only one of these threads could do anything at any point in time. For another thread (or program) to get a chance to do anything the current thread had to relinquish control of the PC back to the operating system. The operating system would then choose which thread would next have a chance to do something. Thus the threads cooperate with each other and the operating system to give each other a chance to do something. This was easy to program, but it meant that programs could behave badly and take full control of the PC locking out other programs.

When Windows NT arrived so did a new style of threading: Preemptive multi-threading. Now threads could run simultaneously, either on separate CPUs or by having the operating system suspend one thread and start another whenever the operating system wanted. This released the power of multi-processor machines and stopped the problem of badly behaved programs. But it was a much harder paradigm to program with.

Threads Pre-Clarion 6

Clarion for Windows was designed to make programming easy. Therefore the problems of preemptive threading were considered unnecessarily complex for the rewards. Clarion threads were not cooperative by their nature. They were standard Win32 threads with explicit synchronization implemented in the RTL to emulate cooperative behavior.

The emulation of cooperative threads made it easy to use variables that were global in scope (can be used anywhere in your program), but whose contents were different depending on what thread was active; for example a global variable with the THREAD attribute. To do this some mechanism was required to make sure that the contents changed each time a different thread became active. Due to the cooperative nature of Clarion threads the RTL was responsible to swap the contents of the threaded variables every time a thread gained or lost focus.

This made it very easy to program, but added the restriction that only one Clarion thread could be active at any time

Threads in Clarion 7

Clarion 7 supports the preemptive thread model. To do this, the automatic data swapping that was done by the RTL had to end. The operating system determines when a thread receives a CPU time slice. So there is no opportunity for the RTL to swap the contents of variables.

Although Clarion now supports preemptive multi-threading, it does not mean you have to suddenly rewrite all your programs. If you follow the guidelines set out in this paper for avoiding the use of global variables whenever possible and using the synchronization techniques described, you’ll be able to deliver new capabilities to your end-users.

Threading Code Design Standards

To allow Clarion to support preemptive threads some changes were required in the language. This was kept to a minimum so as to ease the migration of versions prior to Clarion 6 to Clarion 7.

THREAD attribute on Classes

Prior to Clarion 5.5 if you had a global variable that was an instance of a class the THREAD attribute was ignored. In Clarion 7 any global instance of a class with the THREAD attribute will have separate data for each running thread. The constructor of the class will be called on thread initialization and the destructor will be called on thread termination.

Thread "initialization" and thread "termination" must be defined because it is not always obvious. In general, any DLL has some entry point called by the OS on attaching that DLL to the process, detaching it from the process, on the start and closing a thread. The Clarion RTL translates every such call, to chains of calls to initialize or cleanup data in all modules that have application-wide or thread-wide variables declared. But in certain situations the OS does not do calls to executables, for example, if it is an EXE rather than DLL, or if the DLL is loaded dynamically by the call to LoadLibrary. In the latter case, the OS calls that DLL's entry point for initializing application-wide data only. The CW RTL handles all special cases and calls the code to initialize and also to cleanup threaded data.

The order that class constructors are called is undefined. So do not assume that a different class instance has been constructed when your constructor is called

This feature gives you a lot of power to control things on a thread-by-thread basis. The ABC templates use this strategy to make sure the meta-information about your database is available on each thread.

See the section below "Using hand code" for an example of a global threaded class.

THREAD and EXTERNAL

In versions of Clarion prior to version 6, a variable with the THREAD attribute was allocated a fixed memory location and the runtime library handled swapping the thread specific values of that variable in and out of that memory location. So if you had a file definition in a DLL you would declare it as

```
AFile FILE,DRIVER('TopSpeed'),THREAD
```

Then in your EXE you would declare the file as

```
AFile FILE,DRIVER('TopSpeed'),EXTERNAL
```

With Clarion 7 a variable with the THREAD attribute has different memory allocated for each thread. The compiler generates code to make sure that the right memory is accessed regardless of which thread is running. For the compiler to generate the right code it needs to know if a variable has the THREAD attribute regardless of where the variable is defined. So in Clarion 7 if you have a file definition in a DLL you declare it as

```
AFile FILE,DRIVER('TopSpeed'),THREAD
```

Then in your EXE you declare the file as

```
AFile FILE,DRIVER('TopSpeed'),EXTERNAL, THREAD
```

Notice the THREAD attribute is present in both declarations, unlike previous versions.

ADDRESS of a threaded variable

In version prior to Clarion 6, taking the ADDRESS() of a threaded variable would always return the same result regardless of what thread is running. In Clarion 7 every thread has its own memory location for a threaded variable. So ADDRESS() no longer will return the same value. If you need a constant address for a threaded global variable you can use the new function INSTANCE() and pass a Thread Number of 0. For example, to get a unique identifier for a threaded FILE that you can be certain will be the same regardless of what thread you are on you used to do ADDRESS(MyFile). You now do INSTANCE(MyFile, 0)

LOCKTHREAD, UNLOCKTHREAD and THREADLOCKED

As Clarion 7 works with preemptive threads rather than cooperative threads these functions do nothing by default. However, if you want to work in a cooperative threading environment LOCKTHREAD will call the SYSTEM{PROP:LockThreadHook} function, UNLOCKTHREAD will call the SYSTEM{PROP:UnlockThreadHook} function and THREADLOCKED will call the SYSTEM{PROP:ThreadLockedHook} function.

Coding Techniques for Preemptive Multi-Threading

Don't Use Static Variables

The easiest way to avoid problems is to avoid using static variables. A lot of static variables are declared as static simply because it took less time to use a global or module variable then to pass the data as a parameter. Check your static variables, and if they are used to transfer information between threads pass them as a parameter as needed.

Initialize Once/Read Many

Non Queue static variables

If you have static variables that you use to hold global information that never changes during the running of a program, then you do not need to change them. Provided the variables are initialized in the startup of the program and never changed, there is no need to worry. For example, prior to Clarion 6, when there was no SYSTEM{PROP:DataPath}, it was common to have a file name initialized based on a path stored in an INI file. This practice is still acceptable in a preemptive threading model.

Static Queues that are initialized at startup

A static queue that is initialized at startup and read many times is **not safe**. See the section below on Common Coding Practices that need to change for ideas on how to handle static queues.

Synchronize Access

If you are using static variables as a way to share information between threads, then you will need to synchronize thread access to these variables. The two main ways you will do this is either using a Critical Section Object or a ReaderWriterLocks Object.

If the value to be protected is a LONG (and not in a GROUP) you are probably safe in not synchronizing most thread accesses, since reads/writes happen in only one operation. If a LONG is defined in a GROUP/RECORD/QUEUE then it might not be aligned data and require more then one operation. In summary a LONG has fewer risks of corrupt half updated data then strings, but still have some risks when multiple commands are involved.

See the Synchronization Objects section below for descriptions and examples on how to use these objects.

IMPORTANT!

It is very important that you lock out other threads for as short a time as possible. Ideally, copying the data you need into thread safe variables then releasing the lock. You should not have any user input inside a piece of code that locks other threads.

To guarantee that your static variables are protected it is best to move them into a static class that has read and write methods for manipulating them. Make them PRIVATE, create Get and Set methods, and have synchronization code to protect and manipulate them.

Changes to Common Coding Practices

Using static variables to pass parameters

If you are using a static variable to pass values to a started procedure on a new thread, change the prototype of that procedure so the values it requires are passed as parameters. Both the ABC and Clarion template chains now support this feature.

Using static queues

Static queues that are shared amongst multiple threads will require you to make some code changes with regard to how they are accessed to work in a preemptive threading environment. The problem with queues is that they not only have a queue buffer, but they also have position information. So if one thread reads from the first element of the queue and another thread reads from the second element of the queue you could end up with the following situation:

Thread 1	**Thread 2**
GET(Queue, 1)	
	GET(Queue, 2)
	Assign Queue Data to local var
Assign Queue Data to local var	

Thread 1 gets a time slice from the OS and reads from the Queue, then the OS give thread 2 a time slice before thread 1 makes the assignment of the Queue data to a local variable. Now both thread 1 and thread 2 ends up reading the information for element 2 of the Queue. To avoid this situation you need to synchronize access to the queue's buffer. To keep the time that other threads are locked out while the queue is accessed to a minimum, you should read from the queue and assign the data from the queue's buffer into a threaded memory buffer. See the section below on ICriticalSection for a complete example showing how to do this.

Queues that are only modified at startup

A common coding technique is to set up a static queue that contains data that does not change, for example the states of the USA. This is initialized on program startup, and used throughout the application.

This technique needs to be adjusted to work in a preemptive environment for the reasons stated above. Luckily, the solution is easy -- put a THREAD attribute on the QUEUE

This means that every thread will have its own instance of the QUEUE. The trouble is you need to get the data into that queue for every thread. To accomplish this you create a threaded class to do the copying for you. Below is an example class that will populate a queue for every thread.

In the following example QLock is used to make sure that two threads starting up simultaneously will not interfere with each other. The initial population of the Queue does not need modification provided it is done on the main program thread before any other threads are started. The example shown also makes use of the new INSTANCE() language statement.

```
GlobalQ QUEUE,THREAD
Data     STRING(10)
        END

QLock &ICriticalSection
GlobalQueuePopulator CLASS,THREAD
Construct             PROCEDURE
Destruct              PROCEDURE
                     END
GlobalQueuePopulator.Construct PROCEDURE
BaseQ &GlobalQ
recs  UNSIGNED,AUTO
i     UNSIGNED,AUTO
```

```
    CODE
      IF THREAD() <> 1
        QLock.Wait()
        BaseQ &= INSTANCE(GlobalQ, 1)
        recs = RECORDS(BaseQ)
        LOOP i = 1 TO recs
          GET(BaseQ, i)
          GlobalQ.Data = BaseQ.Data
          ADD(GlobalQ)
        END
        QLock.Release()
      ELSE
        QLock &= NewCriticalSection()
      END

  GlobalQueuePopulator.Destruct PROCEDURE
    CODE
      IF THREAD() = 1
        QLock.Kill()
      END
```

WAIT() function usage

The WAIT function will wait until no other threads want the object. It then takes hold of the object until a subsequent call to Release. Other threads that call the WAIT function will wait indefinitely until the other thread releases the object.

Proper programming techniques are essential here to avoiding a "deadly embrace", or deadlock.

For example, if you execute the following:

```
SyncObj1.Wait
SyncObj2.Wait
```

on one thread and

```
SyncObj2.Wait
SyncObj1.Wait
```

on another thread, you are risking a deadlock.

Deadlocks should not be a big issue if you make sure you have a hierarchy of synchronization objects. Ideally, you only use one object at a time, but if you must use multiple objects, always acquire a lock on the top synchronization object first, then the other one. That way, you can never get into the aforementioned scenario.

Thread Synchronization

The Clarion runtime has a variety of built in interfaces and procedures to help you maintain synchronization between your threads. The POST and EVENT functions have always been in the Clarion language for thread synchronization. SUSPEND and RESUME functions allow you to stop and start another thread, INSTANCE allows you to get another thread's contents for a variable and the ICriticalSection, IMutex, ISemaphore and IReaderWriterLock are interfaces to objects that allow you to synchronize the processing between multiple threads and also multiple processes.

POST/EVENT

You have always been able to synchronize thread processing by posting an event from one thread to another using POST() to send the event and EVENT() to receive it. See the SUSPEND/RESUME section below for an example on using these functions to synchronize two threads.

SUSPEND/RESUME

SUSPEND allows you to stop another process. RESUME starts that process again. You can issue multiple SUSPEND calls for a thread. The same number of RESUME calls must be made for that thread to restart.

The SUSPEND procedure suspends a thread specified by the *threadno* parameter. If the *threadno* parameter is a number of an active thread, its execution is suspended and a suspending counter is incremented. Each additional SUSPEND statement issued to the same active thread will increment the suspending counter by one. Therefore, a thread that has been suspended with a given number of SUSPEND statements can only resume thread execution when an equal number of RESUME statements has been executed.

EXTREME CAUTION should be taken with MDI programs using SUSPEND, as improper use can cause program lockups. All MDI child windows have an MDI client window as a parent, and the MDI client window can send rather than post messages to its child windows.

For example, calling the inter-thread SendMessage modal function causes the calling thread (the MDI client window) to suspend activity until the called thread (the MDI Child window) returns from the call. If the called thread is suspended, we would have a program lockup.

The SUSPEND and RESUME functions can be very useful for controlling threads that are CPU intensive. For example, rebuilding keys on a file. Here is an example program that starts a BUILD of a file and allows the user to pause the build and restart it.

```
          PROGRAM

          MAP

            DoBuild(STRING)

          END


      MyFile FILE,DRIVER('TopSpeed'),PRE(F)

      Key1     KEY(F:Field1),PRIMARY

      Key2     KEY(F:Field2)

      Key3     KEY(F:Field3, Field4)

               RECORD

      Field1     LONG

      Field2     STRING(20)

      Field3     STRING(20)

      Field4     STRING(20)

               END

             END
```

```
BuilderWin WINDOW('Building File'),AT(,,81,22),GRAY
            BUTTON('Suspend Build'),AT(2,3,75,14),USE(?Button)
           END

AllDone  EQUATE(500H)
Building BYTE
ThreadID SIGNED,AUTO

  CODE
    OPEN(BuilderWin)
    ThreadID = START(DoBuild, , THREAD())
    Building = TRUE
    ACCEPT
      CASE EVENT()
      OF AllDone
        MESSAGE('Build Complete')
        BREAK
      OF Event:Accepted
        IF ACCEPTED() = ?Button
          IF Building
            SUSPEND(ThreadID)
            ?Button{PROP:Text} = 'Resume Building'
          ELSE
            RESUME(ThreadID)
            ?Button{PROP:Text} = 'Suspend Build'
          END
        END
      END
    END

DoBuild PROCEDURE (parent)
  CODE
    MyFile{PROP:FullBuild} = TRUE
    BUILD(MyFile)
    POST(AllDone,,parent)
```

INSTANCE

In versions of Clarion prior to Clarion 6.0 a variable's memory location was constant regardless of which thread accessed the variable. Therefore this code would always work:

```
  PROGRAM

  MAP

   AFunc()

  END


GlobVar SIGNED,THREAD

Addr    LONG


  CODE

   Addr = ADDRESS(GlobVar,1)

   START(AFunc)


AFunc PROCEDURE

  CODE

   IF Addr <> ADDRESS(GlobVar)

    MESSAGE('Panic')

   END
```

This sort of code was used in ABFILE.CLW to make sure the file manager matched the file it was meant to manage. To allow programs to know what variable they are really using you can now use the INSTANCE function to get the address of the variable on any thread, and most importantly on thread 1. The above code would need to be modified as follows to work in Clarion 6.0.

```
  PROGRAM
  MAP
   AFunc()
  END

GlobVar SIGNED,THREAD
Addr    LONG

  CODE
   Addr = ADDRESS(GlobVar)
   START(AFunc)

AFunc PROCEDURE
  CODE
   IF Addr <> INSTANCE(GlobVar,0)
    MESSAGE('Panic')
   END
```

Synchronization Objects

A Synchronization Object is an object used to control how multiple threads cooperate in a preemptive environment. There are four Synchronization Objects supplied by the Clarion runtime: Critical Sections (ICriticalSection), Mutexes (IMutex), Semaphores (ISemaphore), and Read/Write Locking (IReaderWriterLock).

Clarion 7 makes it very easy to take advantage of preemptive threads in your applications. All template generated code uses threaded objects to ensure proper behavior. When you embed code that works with threaded data you don't have any worries, but when you access shared non-threaded data you should use a synchronization object.

Note:

Due to the fact that Windows uses procedure-modal methods when dealing with MDI based applications (a program with an APPLICATION window) you must not have any user input when you have control of a synchronization object with an MDI based application. This is likely to lead to your application locking up

If you must have user input, then you must release control of the synchronization object while waiting for user input.

IcriticalSection and CriticalSection

Use an ICriticalSection when you want only one thread accessing some resource (e.g. a global, non-threaded variable) at any one time. An ICriticalSection is faster than an IMutex. If you do not need the extra features of an IMutex, use an IcriticalSection

CriticalSection is a built-in class that allows for easy creation of simple, global synchronization objects.

Following are a couple of examples that make sure that only one thread is accessing a static queue at a time.

```
  PROGRAM

! This program assumes that only WriteToQueue and
! ReadFromQueue directly access NonThreadedQueue
! If other code accesses the queue and does not use
! the QueueLock critical section to synchronize
! access to the queue, then all the work inside WriteToQueue
! and ReadFromQueue is wasted

QueueData GROUP,THREAD
ThreadID    LONG
Information STRING(20)
          END

  !Include CriticalSection
  INCLUDE('CWSYNCHC.INC'),ONCE
  MAP
    WriteToQueue()
    WriteToQueue(*QueueData)
    ReadFromQueue()
    ReadFromQueue(*QueueData)
  END

NonThreadedQueue QUEUE
Data               LIKE(QueueData)
                 END

QueueLock CriticalSection

  CODE

    ! Do everything



WriteToQueue PROCEDURE()
! Assumes QueueData is used to pass data.  This is thread safe
! because QueueData has the THREAD attribute
  CODE
    QueueLock.Wait()  !Lock access to NonThreadedQueue.
    NonThreadedQueue.Data = QueueData
    GET(NonThreadedQueue, NonThreadedQueue.Data.ThreadId)
    IF ERRORCODE()
      ADD(NonThreadedQueue)
    ELSE
      PUT(NonThreadedQueue)
    END
    QueueLock.Release() !Allow other access to the queue

WriteToQueue PROCEDURE(*QueueData in)
  CODE
    QueueLock.Wait()  !Lock access to NonThreadedQueue.
    NonThreadedQueue.Data = in
    GET(NonThreadedQueue, NonThreadedQueue.Data.ThreadId)
    IF ERRORCODE()
      ADD(NonThreadedQueue)
    ELSE
      PUT(NonThreadedQueue)
    END
    QueueLock.Release() !Allow other access to the queue

ReadFromQueue PROCEDURE()
! Returns results in QueueData.  This is thread safe
! because QueueData has the THREAD attribute
  CODE
    QueueLock.Wait()  !Lock access to NonThreadedQueue.
    NonThreadedQueue.Data.ThreadId = THREAD()
    GET(NonThreadedQueue, NonThreadedQueue.Data.ThreadId)
    QueueData = NonThreadedQueue.Data
    QueueLock.Release() !Allow other access to the queue

ReadFromQueue PROCEDURE(*QueueData out)
  CODE
    QueueLock.Wait()  !Lock access to NonThreadedQueue.
    NonThreadedQueue.Data.ThreadId = THREAD()
    GET(NonThreadedQueue, NonThreadedQueue.Data.ThreadId)
    out = NonThreadedQueue.Data
    QueueLock.Release() !Allow other access to the queue
```

The previous example suffers from the problem that anyone can access the global queue and they are not forced to use the matching QueueLock critical section. The following example removes this problem by moving the non-threaded queue and the critical section into a static class.

```
  PROGRAM



QueueData GROUP,THREAD
ThreadID    LONG
Information STRING(20)
          END

NonThreadedQueue QUEUE,TYPE
Data               LIKE(QueueData)
                 END

QueueAccess CLASS
```

```
QueueData &NonThreadedQueue,PRIVATE
QueueLock &ICriticalSection,PRIVATE
Construct     PROCEDURE
Destruct      PROCEDURE
WriteToQueue  PROCEDURE
WriteToQueue  PROCEDURE(*QueueData)
ReadFromQueue PROCEDURE()
ReadFromQueue PROCEDURE(*QueueData)
            END

  INCLUDE('CWSYNCHC.INC')
  MAP
  END

  CODE
    ! Do everything

QueueAccess.Construct PROCEDURE
  CODE
    SELF.QueueLock &= NewCriticalSection()
    SELF.QueueData &= NEW(NonThreadedQueue)

QueueAccess.Destruct PROCEDURE
  CODE
    SELF.QueueLock.Kill()
    DISPOSE(SELF.QueueData)

QueueAccess.WriteToQueue PROCEDURE()
! Assumes QueueData is used to pass data. This is thread safe
! because QueueData has the THREAD attribute
  CODE
    SELF.QueueLock.Wait()  !Lock access to NonThreadedQueue.
    SELF.QueueData.Data = QueueData
    GET(SELF.QueueData, SELF.QueueData.Data.ThreadId)
    IF ERRORCODE()
      ADD(SELF.QueueData)
    ELSE
      PUT(SELF.QueueData)
```

```
          END
          SELF.QueueLock.Release() !Allow other access to the queue

      QueueAccess.WriteToQueue PROCEDURE(*QueueData in)
        CODE
          SELF.QueueLock.Wait()  !Lock access to NonThreadedQueue.
          SELF.QueueData.Data = in
          GET(SELF.QueueData, SELF.QueueData.Data.ThreadId)
          IF ERRORCODE()
            ADD(SELF.QueueData)
          ELSE
            PUT(SELF.QueueData)
          END
          SELF.QueueLock.Release() !Allow other access to the queue

      QueueAccess.ReadFromQueue PROCEDURE()
      ! Returns results in QueueData.  This is thread safe
      ! because QueueData has the THREAD attribute
        CODE
          SELF.QueueLock.Wait()  !Lock access to NonThreadedQueue.
          SELF.QueueData.Data.ThreadId = THREAD()
          GET(SELF.QueueData, SELF.QueueData.Data.ThreadId)
          QueueData = SELF.QueueData.Data
       SELF.QueueLock.Release() !Allow other access to the queue

      QueueAccess.ReadFromQueue PROCEDURE(*QueueData out)
        CODE
         SELF.QueueLock.Wait()  !Lock access to NonThreadedQueue.
         SELF.QueueData.Data.ThreadId = THREAD()
         GET(SELF.QueueData, SELF.QueueData.Data.ThreadId)
         out = SELF.QueueData.Data
         SELF.QueueLock.Release() !Allow other access to the queue
```

IWaitableSyncObject

The IWaitableSyncObject is the base interface for IMutex and ISemaphore. This allows you to create procedures that work with either type of synchronization object without requiring the procedure to know exactly what type of object it is.

IMutex

An IMutex is used when you want to allow only one thread to access a resource. Just like an ICriticalSection. However, IMutexes have the added features of being able to not only synchronize threads, but also synchronize different processes. Thus, if you have a resource that can only have one process accessing it at one time (e.g. a registration file that controls access to multiple programs) then you will need to use an IMutex that is created by calling NewMutex(*Name*). *Name* must be the same for all processes that use it to access the same
set of shared resources.

Another time you would use an IMutex rather than an ICriticalSection is if you do not want to always lock a thread.

Finally, an IMutex works better than an ICriticalSection in MDI applications, as ICriticalSection objects may cause deadlocks. Orphaned mutexes are killed by the operating system after a brief time, but CriticalSections are not. Deadlocks in CriticalSections are usually caused by a programming error (e.g., calling Wait before a form procedure and Release after it).

A Mutex is a very simple way to limit the user to having one instance of your program running at any time. The following example shows the use of the Name parameter for creating a Mutex and the TryWait method to limit your program in this way.

```
PROGRAM
 INCLUDE('CWSYNCHM.INC'),ONCE
 MAP
 END

Limiter &IMutex,AUTO
Result  SIGNED,AUTO
LastErr LONG,AUTO          !<< return error
  CODE
  Limiter &= NewMutex('MyApplicationLimiterMutex',,LastErr)
  IF Limiter &= NULL
   MESSAGE ('ERROR: Mutex can not be created ' & LastErr)
   ELSE
    Result = Limiter.TryWait(50)
     IF Result <= WAIT:OK
      !Do Everything
      Limiter.Release()     !release
     ELSIF Result = WAIT:TIMEOUT
      MESSAGE('Timeout')
     ELSE
      MESSAGE('Waiting is failed ' & Result) !show Result
     END
    Limiter.Kill()
   END
```

The difference between an IMutex and an ISemaphore is an IMutex can only have one thread successfully wait. An ISemaphore can have multiple threads successfully wait. It is also possible to create a semaphore where no thread can successfully wait.

ISemaphore with multiple successful waits

An ISemaphore created with an initial thread count other than zero will allow you to call Wait that number of times before the wait will lock. For example a semaphore created with *MySem &= NewSemaphore(,2)* will allow *MySem.Wait()* to succeed twice without any call to *MySem.Release()*. This is a very easy way to limit the number of threads you have active at any one time.

If at any time you want to allow more calls to Wait to succeed, you can make additional *Release()* calls. The number of extra threads that can be added in this way is limited by the final parameter of *NewSemaphore()*. If you do not want to allow this feature, do not specify the final a maximum.

See the following section for a semaphore that limits the number of threads of a specific type to one.

ISemaphore with no waits

A semaphore created with an initial thread count of 0 will block any call to Wait until a Release is called. You use this type of semaphore to signal another thread that they can do something. For example, signal a thread to send an email to someone because you have sold the last candy bar.

Following is an example where the no wait style of semaphore is used to signal a reader that there is something to read. A multiple successful waits semaphore is used to limit the number of reader threads to 1.

```
   PROGRAM

    INCLUDE('CWSYNCHM.INC')
    INCLUDE('CWSYNCHC.INC')
    INCLUDE('ERRORS.CLW')
    MAP
      Reader()
      Writer()
    END

LogFile FILE,DRIVER('ASCII'),CREATE,NAME('LogFile.txt'),THREAD
          RECORD
Line        STRING(255)
          END
        END

AccessToGlobals CriticalSection
NewData         Semaphore
LimitReaders    &ISemaphore

GlobalStrings   QUEUE,PRE(Q)
Data              STRING(50)
                END
```

```
AppFrame APPLICATION('Reader/Writer'),AT(,,400,240),SYSTEM,MAX,RESIZE
       MENUBAR
         MENU('&File'),USE(?FileMenu)
           ITEM('E&xit'),USE(?Exit),STD(STD:Close)
         END
         MENU('&Launch'),USE(?LaunchMenu)
           ITEM('Reader'),USE(?LaunchReader)
           ITEM('Writer'),USE(?LaunchWriter)
         END
       END
     END

  CODE
    LimitReaders &= NewSemaphore(1)
    SHARE(LogFile)
    IF ERRORCODE() = NoFileErr
      CREATE(LogFile)
      SHARE(LogFile)
    END
    IF ERRORCODE()
      STOP('Log File could not be opened.  Error: ' & ERROR())
    END
    OPEN(AppFrame)
    ACCEPT
      IF EVENT() = EVENT:Accepted
        CASE ACCEPTED()
        OF ?LaunchReader
          START(Reader)
        OF ?LaunchWriter
          START(Writer)
        END
      END
    END
    ! Test to see if Reader is still alive
    IF LimitReaders.TryWait(1) = WAIT:TIMEOUT
      !It is, so lets kill it
      AccessToGlobals.Wait()
      Q:Data = 'exit'
      ADD(GlobalStrings)
      AccessToGlobals.Release()
      NewData.Release() ! Release the reader that is waiting
      LimitReaders.Wait() ! Wait for thread to terminate
      LimitReaders.Release()
    END
    LimitReaders.Kill()

Reader PROCEDURE
  CODE
    ! Check that there are no other readers
    IF LimitReaders.TryWait(1) = WAIT:TIMEOUT
      MESSAGE('Only One Reader Allowed at a time.  ' &|
              'Kill reader by typing ''exit'' in a sender')
      RETURN
    END
    ! If TryWait succeeds, then we have control of
    ! the LimitReaders Semaphore
    SHARE(LogFile)
l1  LOOP
      NewData.Wait() !Wait until a writer signals that there is some data
      AccessToGlobals.Wait()
      LOOP
        GET(GlobalStrings,1)
        IF ERRORCODE() THEN BREAK.
        IF Q:Data = 'exit' THEN
          FREE(GlobalStrings)
          AccessToGlobals.Release()
          BREAK l1
        ELSE
          LogFile.Line = Q:Data
          ADD(LogFile)
        END
        DELETE(GlobalStrings)
      END
      AccessToGlobals.Release()
    END
    CLOSE(LogFile)
    LimitReaders.Release() !Allow a new Reader
    RETURN

Writer PROCEDURE

LocString STRING(50)
Window WINDOW('Writer'),AT(,,143,43),GRAY
       PROMPT('Enter String'),AT(2,9),USE(?Prompt1)
       ENTRY(@s50),AT(45,9,95,10),USE(LocString)
       BUTTON('Send'),AT(2,25,45,14),USE(?Send)
       BUTTON('Close'),AT(95,25,45,14),USE(?Close)
     END

  CODE
    OPEN(Window)
    Window{PROP:Text} = 'Writer ' & THREAD()
    ACCEPT
      IF EVENT() = EVENT:Accepted
        CASE ACCEPTED()
        OF ?Send
          AccessToGlobals.Wait()
          Q:Data = LocString
          ADD(GlobalStrings)
          AccessToGlobals.Release()
          NewData.Release() ! Release the reader that is waiting
        OF ?Close
          BREAK
        END
      END
    END
    RETURN
```

IReaderWriterLock

An IReaderWriterLock can be used to allow multiple threads to read from a global resource, but for only one thread to write to it. No reader is allowed to read the resource if someone is writing and no one can write to the resource if anyone is reading.

An example of this would be where the user specifies screen colors. This is stored in an INI file and read on startup. As the user can change these at any time, the code that changes the values needs to obtain a write lock and all those that read the color information need to obtain a read lock.

Static queues cannot be synchronized via an IReaderWriterLock because reading a queue also modifies its position. All queue access must be considered to cause writes.

Here is some simple code that reads and writes to some static variables. To make sure that no one accesses the statics outside the locking mechanism, the variables are declared as PRIVATE members of a static class.

```
    PROGRAM

    INCLUDE('CWSYNCHM.INC'),ONCE
    MAP
    END

GlobalVars CLASS
AccessToGlobals     &IReaderWriterLock,PRIVATE
BackgroundColor     LONG,PRIVATE
TextSize            SHORT,PRIVATE
Construct           PROCEDURE
Destruct            PROCEDURE
GetBackground       PROCEDURE(),LONG
PutBackground       PROCEDURE(LONG)
GetTextSize         PROCEDURE(),SHORT
PutTextSize         PROCEDURE(SHORT)
           END

  CODE

GlobalVars.Construct           PROCEDURE
  CODE
    SELF.AccessToGlobals &= NewReaderWriterLock()

GlobalVars.Destruct            PROCEDURE
  CODE
    SELF.AccessToGlobals.Kill()

GlobalVars.GetBackground PROCEDURE()
ret SHORT,AUTO
Reader &ISyncObject
  CODE
    ! You need to copy the static variable
    ! to somewhere safe (A local variable) which
    ! can then be returned without fear that
    ! another thread will change it
    Reader &= SELF.AccessToGlobals.Reader()
    Reader.Wait()
    ret = SELF.Background
    Reader.Release()
    RETURN ret

GlobalVars.PutBackground PROCEDURE(SHORT newVal)
Writer &ISyncObject
  CODE
    Writer &= SELF.AccessToGlobals.Writer()
    Writer.Wait()
    SELF.Background = newVal
    Writer.Release()

GlobalVars.GetTextSize         PROCEDURE()
ret SHORT,AUTO
Reader &ISyncObject
  CODE
    ! You need to copy the static variable
    ! to somewhere safe (A local variable) which
    ! can then be returned without fear that
    ! another thread will change it
    Reader &= SELF.AccessToGlobals.Reader()
    Reader.Wait()
    ret = SELF.TextSize
    Reader.Release()
    RETURN ret

GlobalVars.PutTextSize         PROCEDURE(SHORT newVal)
Writer &ISyncObject
  CODE
    Writer &= SELF.AccessToGlobals.Writer()
    Writer.Wait()
    SELF.TextSize = newVal
    Writer.Release()
```

CriticalProcedure

The CriticalProcedure class is a very easy way to use an ISyncObject interface. If you create a local instance of a CriticalProcedure and initialize it, then it will look after the waiting for a lock and releasing the lock on the ISyncObject for you. The main advantage of using the CriticalProcedure class to handle the locking and releasing for you is that if you have multiple RETURN statements in your procedure, you do not have to worry about releasing the lock before each one. The destructor of the CriticalProcedure will handle that for you.

For example, the following code

```
          PROGRAM
          MAP
           WRITETOFILE()
          END

       INCLUDE('CWSYNCHM.INC')
       INCLUDE('CWSYNCHC.INC')



      ERRORFILE FILE,DRIVER('ASCII'),PRE(EF)
      RECORD     RECORD
      LINE        STRING(100)
                 END
                END

      LOCKER &ICRITICALSECTION

          CODE
          Locker &= NewCriticalSection()
          ASSERT(~Locker &= Null)
          !do everything

      WriteToFile PROCEDURE()
        CODE
          Locker.Wait()
          OPEN(ErrorFile)
          IF ERRORCODE()
            Locker.Release()
            RETURN
          END
          SET(ErrorFile)
          NEXT(ErrorFile)
          IF ERRORCODE()
            Locker.Release()
            RETURN
          END
          EF:Line = 'Something'
          PUT(ErrorFile)
          CLOSE(ErrorFile)
          Locker.Release()
```

Can be shortened, and made less error prone, to this:

```
          PROGRAM
          MAP
           WRITETOFILE()
          END

      INCLUDE('CWSYNCHM.INC')
      INCLUDE('CWSYNCHC.INC')

      ERRORFILE FILE,DRIVER('ASCII'),PRE(EF)
      RECORD     RECORD
      LINE        STRING(12)
                 END
                END

      LOCKER &ICRITICALSECTION

          CODE
          Locker &= NewCriticalSection()
          ASSERT(~Locker &= Null)
          !do everything

      WriteToFile PROCEDURE()

      CP CriticalProcedure

        CODE
          CP.Init(Locker)
          OPEN(ErrorFile)
          IF ERRORCODE()
            RETURN
          END
          SET(ErrorFile)
          NEXT(ErrorFile)
          IF ERRORCODE()
            RETURN
          END
          EF:Line = 'Something'
          PUT(ErrorFile)
          CLOSE(ErrorFile)
```

A second feature of the CriticalProcedure is that it allows you to RETURN a global (or other non-threaded static) variable from a prototype that requires protection. The return value will be pushed onto the return stack, and then during cleanup the Destructor will release the SyncObject. Another other way to RETURN a global string is to copy it to a local variable, release the SyncObject and RETURN the local string.

Note:

Since the base synchronization objects (CriticalSection and Mutex) support nested calls

(wait-wait-release-release), the operating system keeps a wait count, and you must RELEASE as

many times as you WAIT. The CriticalProcedure does not support nested calls (e.g., Init-Init-Kill-Kill). The second call to CriticalProcedure.Init will release the currently locked object (with a CP.Kill), and then try to lock the newly named object. There is no check made the exact same object is being "relocked".

So, this will work with a Critical SECTION:

```
CrSec  &ICriticalSection
  ...
  CrSec.Wait
    ....
    CrSec.Wait
       ... release count now 2 ..
    CrSec.Release
    ... release count 1 ..
    .. code that needs protection ..
  CrSec.Release
```

Similar code with a Critical PROCEDURE will not

```
CrPro  CriticalProcedure
  ...
  CrPro.Init(CrSec)
   ....
   CrPro.Init(CrSec)!CrSec is Released and Relocked so could lose the lock here
    ... release count is 1 and not 2 ..
    CrPro.Kill      !Lock totally lost here
     ... code NOT proctected ..
 CrPro.Kill         !pointless, don't own it now
```

Summary of Synchronization Objects Properties and Methods

The following section provides a quick summary description of the common properties and methods used with the Synchronization Objects discussed previously.

Prototypes

Note: The following prototypes can be found in **CWSYNCHM.INC**

GetMutex

GetSemaphore

NewCriticalSection

NewMutex

NewNamedSemaphore

NewReaderWriterLock

NewSemaphore

GetMutex

GetMutex(*name*, <*error*>)

name	A string constant or variable that names the **IMutex** object.
Error	A LONG variable that returns any operating system error.

Purpose:
Returns a reference to an IMutex for a Mutex that has already been created with NewMutex( )

If the Mutex has not been previously created, a NULL is returned.

Example:

```
MySem    &IMutex
   CODE
   MySem &= GetMutex('MyApp_RequestServer')
   IF MySem &= NULL
     MESSAGE('Server not running')
     RETURN(False)
   END
```

GetSemaphore

GetSemaphore(*name*, <*error*>)

name	A string constant or variable that names the **ISemaphore** object.
Err	A LONG variable that returns any operating system error.

Purpose:
Returns a reference to an ISemaphore for a semaphore that has already been created with NewNamedSemaphore( )

If the semaphore has not been previously created, a NULL is returned.

Example:

```
MySem    &ISemaphore
   CODE
   MySem &= GetSemaphore('MyApp_RequestServer')
   IF MySem &= NULL
       MESSAGE('Server not running')
      RETURN(False)
   END
```

NewCriticalSection

NewCriticalSection ()

Purpose:
Returns a reference to a new ICriticalSection

Example:

```
! This code is invoked for every started thread before call to ThreadProc.
! Depending on the current thread, Action.Construct executes one of 3
! handler procedures.

Action.Construct  PROCEDURE()
  CODE
  SELF.Threadno = THREAD()

  CASE SELF.ThreadNo
```

```
OF 1
  Sync &= NewCriticalSection()          ! Create the critical section object
  MainThread &= SELF                    ! Set MainThread to instance of Action
                                        ! for the main thread
  FrameProc (SELF)
  MainThread &= NULL
    Sync.Kill()                         ! Delete the critical section
OF 2
  ToolThread &= SELF                    ! Set ToolThread to instance of Action
                                        ! for the toolbox's thread
  ToolProc (SELF)
  ToolThread &= NULL
ELSE
  ChildProc (SELF)
END
RETURN
```

NewMutex

NewMutex( ),
NewMutex(*name*, *owner*, *<error>***)**

name	A string constant or variable that names the new IMutex object. If a *name* parameter is not supplied, **NewMutex** is used for synchronizing threads only. If a *name* is supplied, NewMutex can be used to synchronize multiple processes rather than just the threads within a process.
owner	A BYTE variable or constant that specifies the initial owner of the Mutex. If this value is TRUE, and the caller creates the Mutex, the calling thread obtains ownership of the Mutex. This is the equivalent of calling **Mutex.Wait** immediately after **NewMutex**(). If the caller did not create the mutex, then the calling thread does not obtain ownership of the Mutex. To determine if the caller created the Mutex, you need to check the value of *Err*
error	A LONG variable or constant that returns any operating system error. ERROR_ALREADY_EXISTS (183) indicates that the caller did not create the Mutex (a handle is still returned but owner is ignored) because another process or thread has already created the Mutex.

Purpose:
Returns a reference to a new IMutex. If the Mutex could not be created, a Null value will be returned. Check the *Error*

parameter for the reason. (Some reasons for failure can be that another object (e.g., semaphore) exists with the same name, or a different user has created the object (e.g., security error)).

NewNamedSemaphore

NewNamedSemaphore (*name*, *initial*, *max* , <*error*>)

name	A string constant or variable that names the new **ISemaphore** object.
initial	A LONG variable or constant that indicates the maximum number of threads that can access the semaphore. The default value is zero.
max	A LONG variable or constant that indicates the maximum number of threads that can simultaneously access the semaphore.
error	A LONG variable that returns any operating system error. An ERROR_ALREADY_EXISTS (183) error indicates that another process or thread has already created the Mutex.

Purpose:
Returns a reference to a new ISemaphore. This semaphore can be used for synchronizing threads or processes. If an ERROR_ALREADY_EXISTS (183) is posted to *error*, then the Semaphore already existed, and *initial* and *max* are ignored.

Using the default settings, NewNamedSemaphore will create a semaphore that no threads can wait on until someone calls a Release.

NewReaderWriterLock

NewReaderWriterLock (*WritersHavePriority*)

WritersHavePriority	A BYTE variable or constant that sets priority of the Writer.

Purpose:
Returns a reference to a new IreaderWriterLock

If *WritersHavePriority* is TRUE, a Writer waiting for ownership of the ReaderWriterLock object will take priority over any Readers that are waiting for ownership.

If *WritersHavePriority* is FALSE, all Readers waiting for ownership of the ReaderWriterLock object will take priority over any Writer waiting for ownership.

NewSemaphore

NewSemaphore (*initial* , *max*)

Initial	A LONG variable or constant that indicates the maximum number of threads that can access the semaphore. The default value is zero.
Max	A LONG variable or constant that indicates the maximum number of times that the semaphore can be owned. It is a maximum resource count.

Purpose:
Returns a reference to a new ISemaphore

If initial is non-zero, then this indicates how many threads can initially and simultaneously have access to the semaphore

max indicates the maximum number of threads that can simultaneously access the semaphore. By default, this will create a semaphore that no threads can wait on until a Release is called.

Synchronization Object Interface Methods Summary

ICriticalSection.Wait( ) and IMutex.Wait( )

Implementation:

The first time either method is called, the calling thread gains control of the synchronization object. Any other threads that call Wait will wait until **Release** is called.

A thread can call **Wait** multiple times. **Release**() must be called once for each call to **Wait**.

ICriticalSection.Release( ) and IMutex.Release(*count*)

Implementation:
Each method decrements the counter of **Wait** calls. When the **Wait** count reaches zero, the synchronization object is released for any other thread to acquire control with a call to **Wait**.

Count specifies to release a given number of times up to the number of previously successful wait calls.

Kill( )

Implementation:
Releases all resources allocated when the target synchronization object was created with the appropriate **NEW** function.

ISemaphore.Wait( )

Implementation:

This method will wait until shared control of the synchronization object is allowed. Once this occurs, Wait will take partial control of the synchronization object.

A thread can call **Wait( )** multiple times. **Release( )** must be called once for each call to Wait.

ISemaphore.TryWait (ms) and IMutex.TryWait (ms)

TryWait PROCEDURE(LONG milliseconds),SIGNED,PROC

Implementation:

Attempts to gain control of the synchronization object within the *ms* parameter time in milliseconds. Return value is a signed EQUATE with one of the following results:

WAIT:OK
Control was acquired

WAIT:TIMEOUT
Control could not be acquired within the time limit

WAIT:NOHANDLE
Something is seriously wrong

WAIT:FAILED
Something else is seriously wrong

WAIT:ABANDONED
Another thread that had control of the synchronization object has ended without calling release. The caller now has the control the same way as if a WAIT:OK was returned. This is more of a warning indicating a flaw in the program logic.

ISemaphore.Release( )

Implementation:
Decrements the counter of **Wait()** calls. When the **Wait()** count reaches zero, the synchronization object is released for any other thread to acquire control with a call to **Wait()**.

If more **Release()** calls are made than **Wait()** calls, the total number of threads allowed to control the semaphore is incremented up to the maximum parameter of *NewSemaphore*.

Count specifies to release a given number of times up to the number of previously successful wait calls.

ISemaphore.Handle( ) and IMutex.Handle( )

For internal use only

IReaderWriterLock.Reader( )

Purpose:
Returns a synchronization object that can be used to lock out writers. Any successful call to Reader.Wait() will stop any attempt at Writer.Wait() until Reader.Release() is called.

IReaderWriterLock.Writer( )

Purpose:
Returns a synchronization object that can be used to lock out any other writer and all readers. Any successful call to Writer.Wait() will stop any attempt at Writer.Wait() and Reader.Wait() until Writer.Release() is called

CriticalProcedure.Init (syncObj)

Purpose:
Calls syncObj.Wait(). syncObj.Release is called by the destructor.

Index