Documentation Highlights
for Clarion 6.1

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

Printed in the United States of America (0504)

Contents:

Clarion 6.1 Documentation Highlights

This document contains updates that were made to the help files and documentation since the release of Clarion 6 Gold. It is divided by category for easy reference.

This document intends to highlight important changes in Clarion 6.1, and is not intended to be a comprehensive reference of every feature and change. For that information, please refer to the ReadMe file included with the Clarion 6.1 installation.

ABC Library Reference

IReportGenerator Interface:

OpenPage (begin a page print)

OpenPage(*left***,** *top***,** *right***,** *bottom***,** *pagename***), BYTE**

OpenPage	Called before each page is printed to detect if an error has occurred.
left *top* *right* *bottom*	A SHORT for each parameter identifying the respective boundaries of the page.
pagename	A STRING variable or constant that identifies the name of the WMF file to be printed.

The **OpenPage** method is used to detect an error level state that may exist before each page (WMF file) has been printed. If the page is ready to be printed, **OpenPage** returns Level:Benign.

Implementation: Called immediately before each WMF page is printed.

Return Data Type: BYTE

ProcessString (print a string control)

ProcessString (*StringFormatGrp str*grp, text, comment* **)**

ProcessString	Prints a STRING control to a target output document.
strgrp	A TYPEd GROUP structure that holds all of the properties of the target STRING control.
text	A string constant, variable, EQUATE, or expression containing the STRING contents.
comment	A string constant, variable, EQUATE, or expression containing information necessary for the STRING control to be properly rendered to the target document.

The **ProcessString** method prints a STRING control to the appropriate document format. The *comment* parameter is used to send the appropriate formatting information to the target document type, and is limited to 2056 characters. (See)

Implementation: The *strgrp* group contains the position, alignment, styles, character set and other attributes of a STRING control from the contents of the passed *StringFormatGrp*.

ProcessText (print a text control)

ProcessText (TextFormatQueue txtque*, comment* **)**

ProcessText	Prints a TEXT control to a target output document.
txtque	A QUEUE structure that holds all of the properties of each line of the target TEXT control, and its contents.
comment	A string constant, variable, EQUATE, or expression containing information necessary for each line of the TEXT control to be properly rendered to the target document.

The **ProcessText** method prints a TEXT control to the appropriate document format. The *comment* parameter is used to send the appropriate formatting information to the target document type, and is limited to 2056 characters. (See COMMENT)

Implementation: Each *txtque* QUEUE entry contains the position, alignment, styles, character set and other attributes of the TEXT control.

TransactionManager Overview

The TransactionManager class is used to manage a transaction processing “frame”. It wraps all of the classic operations normally used in a typical transaction process, including LOGOUT, COMMIT, and ROLLBACK operations, and allows you to control them through a simple set of methods. Nearly all ISAM and SQL tables support transaction processing. Please refer to the Database Drivers Help topic for more specific information regarding each individual driver.

TransactionManager Concepts

Override and Control of one or all of the Template-Based Transaction Frames

In a standard application created in the Application Generator, transaction processing of data elements is handled by the target RelationManagers for each primary table. The ABC Templates set the RelationManager’s UseLogout property based on the **Enclose RI code in transaction frame** check box in the *Global Properties* dialog. You can use the TransactionManager (with the help of the supporting templates) to turn off the RelationManager support for transaction framing, and specifically customize the tables that you need to enclose in a transaction frame in a target Form or Process procedure.

Simplified Custom Transaction Processing

All of your hand-coded transaction processing frames can now be encapsulated in the TransactionManager. Using its available methods ensures that proper initialization, processing, and error checking will be performed.

TransactionManager ABC Template Implementation

The TransactionManager is supported by two Extension templates. The Save Button Transaction Frame extension template is used to control transaction processing via the TransactionManager in any Form (update) procedure that uses the Save Button control template. Using the Process Transaction Frame extension template, you can override and control any needed transaction processing that needs to be applied in any process procedure.

The Process Transaction Frame Checkpoint code template is used in any process procedure to specify and control transaction processing over a specific batch of records instead of the normal default processing of individual records.

TransactionManager Relationship to Other Application Builder Classes

The TransactionManager is closely integrated with RelationManager objects. These objects are added to a protected TransactionManagerQueue, where a reference to each RelationManager object and thread instance is stored.

TransactionManager Source Files

The TransactionManager source code is installed by default to the Clarion \LIBSRC folder. The TransactionManager source code and its respective components are contained in:

ABFILE.INC	TransactionManager declarations
ABFILE.CLW	TransactionManager method definitions

TransactionManager Conceptual Example

The following examples show a typical sequence of statements to declare, instantiate, initialize, use, and terminate a TransactionManager.

```
!This example shows all that is needed to implement transaction processing
!using the TransactionManager

MyTransaction  TransactionManager
ReturnValue BYTE
 CODE
  Relate:Invoice.Open()
  MyTransaction.AddItem(Relate:Invoice)
  MyTransaction.AddItem(Relate:Items)
  ReturnValue = Level:Benign
  IF MyTransaction.Start()=Level:Benign !Initialize, begin LOGOUT
     !......
 !Work with the Tables here and set the ReturnValue to Level:Fatal
 !if there are any errors.
     !......
     MyTransaction.Finish(ReturnValue) !Commit or rollback based on errorlevel
  END
!-----------------------------------------------------------------
!This next partial code example demonstrates how to execute the transaction
!in a process. It uses the SetLogoutOff\RestoreLogout methods. This is used
!when you need a longer transaction in a process where you don't need to
!continually set the Uselogout to TRUE/FALSE repeated times (via Start)

...........
   MyTransaction.AddItem(Relate:Invoice)
   MyTransaction.AddItem(Relate:Items)
   MyTransaction.SetLogoutOff()
...........
   ReturnValue = MyTransaction.Start()
...........
...........
   MyTransaction.Finish(ReturnValue)
   ReturnValue = MyTransaction.Start()
...........
...........
   MyTransaction.Finish(ReturnValue)
...........
   MyTransaction.RestoreLogout()
...........
! ----------------------------------------------------------------
!This example shows the use of the Process virtual method.
  PROGRAM
  MAP.
MyTransaction  CLASS(TransactionManager)
Process         PROCEDURE(),BYTE,VIRTUAL
               END
```

```
ReturnValue BYTE
 CODE
   Relate:Invoice.Open()
   MyTransaction.AddItem(Relate:Invoice)
   MyTransaction.AddItem(Relate:Items)
   MyTransaction.Run()!This will all the work for you

MyTransaction.Process   PROCEDURE()
ReturnValue   BYTE
 CODE
  ReturnValue = Level:Benign
  !Work with the Tables here and set the ReturnValue to Level:Fatal
  !if there are any errors.
 RETURN ReturnValue
```

TransactionManager Properties

The TransactionManager class contains no public properties.

TransactionManager Methods

AddItem (add a RelationManager to transaction list)

Finish (rollback or commit transaction)

Process (a virtual to process transaction)

Reset (remove all RelationManagers from transaction list)

RestoreLogout (restore all RelationManagers in transaction list to previous status)

Run (initiates transaction sequence)

SetLogoutOff (turn off logout for all RelationManagers in transaction list)

SetTimeout (set timeout used in transaction)

Start (start the transaction)

TransactionCommit (commit the transaction)

TransactionRollBack (rollback the transaction)

AddItem (add a RelationManager to transaction list)

AddItem(*RM,cascadechildren* **)**

AddItem	Add the RelationManager object to the TransactionManager list queue.
RM	The label of the RelationManager object.
Cascadechildren	An integer constant, variable, EQUATE, or expression that indicates whether the TransactionManager automatically includes any child tables defined by the Relationmanager object into the transaction processA value of one (1 or True) automatically includes all child tables; a value of zero (0 or False) excludes all child tables. If omitted, *cascadechildren* defaults to 1.

AddItem adds a reference to a RelationManager object to the TransactionManager's protected TransactionManagerQueue. This, in effect adds (by default), all tables defined in the RelationManager object to the processing of the TransactionManager.

Implementation: To include a primary table and its associated children in an impending TransactionManager process, you should call AddItem and specify the appropriate RelationManager object in the Init method of the WindowManager or ProcessManager.

Example:

```
IF SELF.Request<>ViewRecord  !for any update
 !activate the Roysched table, but not its defined child tables
 Transaction.AddItem(Relate:Roysched,False)
 !but include the Titles table specifically
 Transaction.AddItem(Relate:Titles,True)
END
```

See Also: RelationManager.UseLogout

Finish (rollback or commit transaction)

Finish(*errorlevel* **)**

Finish	Completes the transaction processing
errorlevel	An integer constant, variable, EQUATE, or expression that sets the current error level, and determines the success of the transaction process.

Finish completes the TransactionManager process. Using the *errorlevel* value it will rollback or commit the transaction. An *errorlevel* of Level:Benign will commit (complete) the transaction, where any other *errorlevel* set will force a rollback (cancellation) of the transaction.

Implementation: The **Finish** method should be called in the TakeCompleted method to validate a transaction. During a process, it can be called at any time to commit or rollback a batch of records processed. The method calls either the TransactionCommit or TransactionRollback methods in order to complete the transaction process.

Example:

```
ReturnValue = PARENT.TakeCompleted()
! A ReturnValue other than Level:Benign will rollback the transaction
IF SELF.Request<>ViewRecord
   Transaction.Finish(ReturnValue)
END
  RETURN ReturnValue
END
```

See Also: Start, TransactionCommit, TransactionRollback

Process (a virtual to process transaction)

Process( ),BYTE,VIRTUAL

Process	Process any data during the transaction process.

Process is a virtual placeholder method used to work with any tables affected by the transaction process, and can return the correct error level to control if the transaction should be completed or aborted (COMMIT or ROLLBACK respectively).

Return Value: BYTE

Implementation: The Process method is a virtual method that will be called from the Run method only if the Start method first returns a Level:Benign error level. After that, if the Process method returns any error level other than Level:Benign, the transaction will rollback.

Example:

```
MyTransaction.Process   PROCEDURE()
ReturnValue   BYTE
 CODE
    ReturnValue = Level:Benign
    !Work with the Tables and set the ReturnValue
    !to Level:Fatal here if there are any errors.
    RETURN ReturnValue
```

Reset (remove all RelationManagers from transaction list)

Reset( )

Reset	Remove all entries from the TransactionManager list queue.

Reset is used to remove all RelationManager objects that have been added to the TransactionManager's protected TransactionManagerQueue. In effect, the queue is freed, and any impending transaction processing will not occur. If a transaction is already in progress, the **Reset** method is ignored.

Implementation: Use the Reset method at any time prior to starting a transaction process if you need to cancel the entire operation for any reason.

RestoreLogout (restore all RelationManagers in transaction list to previous logout status)

RestoreLogout( )

RestoreLogout	Restores all RelationManager objects in the TransactionManager list queue to their original transaction status.

RestoreLogout will restore the respective UseLogout property set in each RelationManager involved in the transaction to its previous status.

Implementation:	The Init method of the respective RelationManager sets the value of the UseLogout property. The ABC Templates set the UseLogout property based on the **Enclose RI code in transaction frame** check box in the **Global Properties** dialog. RestoreLogout sets this property to its previous status. Normally, this follows a call to the SetLogoutOff method.

See Also: UseLogout, SetLogoutOff

Run (initiates transaction sequence)

Run(*timeout* **)**

Run	Initiates the transaction sequence.
timeout	A numeric constant or variable specifying the number of seconds to attempt to begin the transaction for files contained in the target RelationManager objects before aborting the transaction and posting an error.

Run is used to initiate the TransactionManager transaction process. If the *timeout* value is not exceeded, the Start\Process\Finish-TransactionCommit or TransactionRollBack methods will be subsequently called.

Implementation: The Run method is not used by the ABC template chain. It is a method provided for developers who are writing custom source code using the TransactionManager.

SetLogoutOff (turn off logout for all RelationManagers in transaction list)

SetLogoutOff( )

SetLogoutOff	Turn off default logout setting in all RelationManagers stored in TransactionManager list queue

SetLogoutOff is used to set the default logout setting in the appropriate RelationManager objects contained in the protected TransactionManagerQueue to OFF. This allows the TranactionManager to control the tranaction process through its own properties and methods.

Implementation: The SetLogoutOff method loops through the list of RelationManager objects listed by the TranactionManager, saves the appropriate status of the RelationManager's UseLogout property, and sets the UseLogout property to FALSE. It is internally called by the Start method, or may be called explicitly in a process where multiple transactions with batches of records may occur, and the continued call to the Start method for each batch does not need to continually reset the UseLogout property.

Example:

```
TransactionManager.Start        PROCEDURE()
I       LONG,AUTO
RetVal  BYTE,AUTO
 CODE
    IF SELF.TransactionRunning THEN RETURN Level:Fatal.
    IF SELF.AutoLogoutOff
       FREE(SELF.UselogoutList)
       SELF.LogoutOff = True
    END
    FREE(SELF.RMList)
    LOOP I=1 TO RECORDS(SELF.Files)
         GET(SELF.Files,I)
         IF NOT ERRORCODE()
            IF SELF.AutoLogoutOff
               SELF.SetLogoutOff(SELF.Files.RM)
            END
            RetVal = SELF.AddFileToLogout(SELF.Files.RM,SELF.Files.Cascade)
            IF RetVal<>Level:Benign
               BREAK
            END
         END
    END
```

SetTimeout (set timeout used in transaction)

SetTimeout(*seconds* **)**

SetTimout	Sets the TransactionManager's LOGOUT timeout value.
timeout	A numeric constant or variable specifying the number of seconds to attempt to begin the transaction for files contained in the target RelationManager objects before aborting the transaction and posting an error.

SetTimeout is used to set the TransactionManager's LOGOUT timeout value. The internal default value is 2 seconds.

Start (start the transaction)

Start( ),BYTE,VIRTUAL

Start	Begin the transaction process.

Start is a virtual method used to begin the TransactionManager transaction process. **Start** makes sure that a transaction is not already running, clears the target RelationManager's internal UseLogout property, manages and issues a LOGOUT for all active tables contained in the target RelationManagers maintained by the TransactionManager.

If the initialization and LOGOUT statement are successful, **Start** returns a Level:Benign error level. If the **Start** method is for any reason unsuccessful, a Level:Fatal error level is returned.

Return Value: BYTE

Implementation: In a form (update) procedure, the **Start** method is called just prior to the Window Manager's TakeCompleted method. In a process procedure that implements the TransactionManager, the Start method can be called for each individual method processed, or can be called for a specified number of records processed.

Example:

```
!In a Form procedure
  IF SELF.Request<>ViewRecord
   ReturnValue = Transaction.Start()
   IF ReturnValue<>Level:Benign THEN RETURN ReturnValue.
  END
  ReturnValue = PARENT.TakeCompleted()
  ! A ReturnValue other than Level:Benign will rollback the transaction
  IF SELF.Request<>ViewRecord
   Transaction.Finish(ReturnValue)
  END

!In a process procedure that individually processes each record
ThisWindow.OpenReport PROCEDURE

ReturnValue          BYTE,AUTO

  CODE
  ReturnValue = PARENT.OpenReport()
  IF ReturnValue = Level:Benign
     ReturnValue = Transaction.Start()
  END
  RETURN ReturnValue
```

```
!In a process procedure that processes a batch of records
ThisProcess.TakeRecord PROCEDURE

ReturnValue          BYTE,AUTO

  CODE
  ReturnValue = PARENT.TakeRecord()
  ! ------------------------------------------------
  !
  ! Transaction Check Point
  ! The transaction will be saved till this point
  ! and a new one will be started
  !
  IF SELF.RecordsProcessed % 100 = 0
     Transaction.Finish(ReturnValue)
     IF ReturnValue = Level:Benign
        ReturnValue = Transaction.Start()
     END
  END
  ! ------------------------------------------------
  PUT(Process:View)
  IF ERRORCODE()
    GlobalErrors.ThrowFile(Msg:PutFailed,'Process:View')
    ThisWindow.Response = RequestCompleted
    ReturnValue = Level:Fatal
  END

  RETURN ReturnValue
```

TransactionCommit (commit the transaction)

TransactionCommit( ),VIRTUAL

TransactionCommit Commit (complete) the transaction process.

TransactionCommit is a virtual method used to complete the TransactionManager's transaction process by issuing a COMMIT statement. In addition, the UseLogout property of each RelationManager used in the transaction is restored to its previous state. It also checks to make sure if the transaction has already been completed.

Implementation: The **TransactionCommit** method is called by the Finish method if a Level:Benign error level has been posted.

Example:

```
TransactionManager.Finish       PROCEDURE(BYTE pErrorLevel)
 CODE
  IF NOT SELF.TransactionRunning THEN RETURN.
   IF pErrorLevel = Level:Benign
    SELF.TransactionCommit()
   ELSE
    SELF.TransactionRollBack()
   END
```

See Also: Finish, TransactionRollback, COMMIT

TransactionRollBack (rollback the transaction)

TransactionRollback( ),VIRTUAL

TransactionRollback Rollback (abort) the transaction process.

TransactionRollback is a virtual method used to abort the TransactionManager's transaction process by issuing a ROLLBACK statement. In addition, the UseLogout property of each RelationManager used in the transaction is restored to its previous state. It also checks to make sure if the transaction has already been completed.

Implementation: The **TransactionRollback** method is called by the Finish method if a Level:Fatal error level has been posted.

Example:

```
TransactionManager.Finish       PROCEDURE(BYTE pErrorLevel)
 CODE
  IF NOT SELF.TransactionRunning THEN RETURN.
   IF pErrorLevel = Level:Benign
    SELF.TransactionCommit()
   ELSE
    SELF.TransactionRollBack()
   END
```

See Also: Finish, TransactionCommit, ROLLBACK

Window Manager – Properties

MyWindow (the Managed WINDOW)

MyWindow &WINDOW

The **MyWindow** property is a reference to the managed primary WINDOW structure. The WindowManager uses this property to open the WINDOW.

Implementation: The Open method sets the MyWindow property.

OwnerWindow (the Managed owner WINDOW)

OwnerWindow &WINDOW

The **OwnerWindow** property is a reference to the managed owner WINDOW structure. The WindowManager uses this property to associate the owner with an opened WINDOW.

Implementation: The Open method sets the OwnerWindow property.

Window Manager – Methods

Open (open and initialize a window structure)

Open, VIRTUAL
Open (*mainwindow, <ownerwindow>*), **VIRTUAL**

mainwindow The label of the window that needs to be opened.

ownerwindow The label of the owner window, if applicable. This is the label of the APPLICATION or WINDOW structure which "owns" the *mainwindow* being opened.

The **Open** method, when called with the *mainwindow* and optional *ownerwindow* parameters, is used to open a window for processing. A *mainwindow* with an *ownerwindow* always appears on top, and is automatically hidden if the *ownerwindow* is minimized or hidden. If the *ownerwindow* is closed, all owned windows are also automatically closed.

The **Open** method when called without parameters, prepares the window for display. It is designed to execute on window opening events such as EVENT:OpenWindow and EVENT:GainFocus, and can optionally translate a window if necessary.

Implementation: The Open method invokes the Translator if present and calls the Reset method to reset the WINDOW. The TakeWindowEvent method calls the Open method.

Example:

```
ThisWindow.TakeWindowEvent  PROCEDURE
  CODE
  CASE EVENT()
    OF EVENT:OpenWindow
      IF ~BAND(SELF.Inited,1)
        SELF.Open(Window)
      END
    OF EVENT:GainFocus
      IF BAND(SELF.Inited,1)
        SELF.Reset
      ELSE
        SELF.Open
      END
  END
  RETURN Level:Benign

ThisWindow.Open  PROCEDURE
  CODE
    IF ~SELF.Translator&=NULL
      SELF.Translator.TranslateWindow
    END
    SELF.Reset
    SELF.Inited = BOR(SELF.Inited,1)

!Usage with parameter(s):
 SELF.Open(Window)
```

TakeNotify (a virtual to process EVENT:Notify)

TakeNotify (*notifycode*, *thread***,** *parameter* **),VIRTUAL, PROC**

notifycode an UNSIGNED variable that receives a notify code value passed by the sender with a NOTIFY statement.

thread an optional SIGNED variable that receives the number of the sender's thread parameter.

parameter a LONG variable that receives the parameter passed by the sender with a NOTIFY statement.

TakeNotify is a virtual method used to process valid EVENT:Notify events for the window's controls and returns a *Level:Benign* value by default. This method is called if EVENT:Notify is received by the window, and the NOTIFICATION function (and subsequently this method) returns TRUE if the *parameter* values match the values from the NOTIFY function that posted the event.

Implementation: TakeNotify is called by the TakeWindowEvent method if a valid notification is detected.

Return Data Type: BYTE

Example:

```
WindowManager.TakeWindowEvent      PROCEDURE
RVal BYTE(Level:Benign)
NotifyCode      UNSIGNED
NotifyThread    SIGNED
NotifyParameter LONG
  CODE
  CASE EVENT()
  OF EVENT:Notify
    IF NOTIFICATION(NotifyCode,NotifyThread,NotifyParameter)
      RVal = SELF.TakeNotify(NotifyCode,NotifyThread,NotifyParameter)
    END
  END
```

See Also: NOTIFICATION, NOTIFY

Overview of ErrorClass changes in Clarion 6.1

In Clarion version 5.5 and prior, the Error Class was designed to be a Global class, using just one instance of the class in a program (EXE or EXE with multi DLL) that would be used by the entire application. This was possible because the previous thread model did not allow two different threads to use the global ErrorClass at the same time. With the incorporation of the new thread model in Clarion 6, this limitation disappears, and it is now possible that the global ErrorClass can be used by two different threads at the same time.

In the first release (Clarion 6.0), the ErrorClass was changed to a THREADed Class. This change made it safe to be used in a preemptive thread environment.

There are other possible designs that could have been used. Another approach that could have been taken would be to add thread synchronization to the class. Yet another design approach is to divide the class into two classes: one class contains thread dependent data and the other contains thread independent data. This is the design approach that has been implemented in version 6.1.

If a class has some data that needs to be thread specific and other data that does not, there are several design options that need to be considered.

For thread independent data, a solution is to add some kind of synchronization to the class (e.g., CriticalProcedure) in order to prevent two different threads from accessing these values at the same time. Of course, care must be used in that the data and scope where we use it should maintain this synchronization.

For the thread dependent data, one solution is to move the threaded data to a synchronized queue that stores the data, using the thread number as the queue's key. Another solution (and the one used with Clarion 6.1) is to create a *new* class (ErrorStatusClass) that is specifically used as a *container* for the thread dependent data. This second option is equivalent to working with only *one* class (the ErrorClass) that has threaded *and* non-threaded parts. The thread dependent part will create and destroy a new instance for each thread and the thread independent part stores the "thread independent ID" of the thread dependent part so it can use it with the associated thread number in order to get the correct reference to the threaded class for a specific thread.

Because any access to the threaded class parts will need to be done using some function that first retrieves the correct class reference, the changes made in the ErrorClass for version 6.1 emulates this implementation, where all access to the key property attributes are now done through an associated pair of GET*propertyname* and SET*propertyname* methods, where *propertyname* is the property that is affected.

Sometimes these GET/SET methods are used to wrap the synchronized object, and other times they control the access to the specific threaded class' properties or queues.

Also, these changes to the ErrorClass simplify its use in multi DLL applications, because the global ErrorClass is consistent for each thread, and synchronization is straightforward.

Also, this change maintains the ability to customize errors in only one place and use them throughout the application. A single queue is used to store the error list, so extra memory is not required by additional threads. All methods are declared in the non-threaded class, so only a single instance of these methods is loaded in memory. Access to the global class is implemented in such a way that the addition of the synchronization methods will not slow down the application's performance.

ErrorClass – Properties

As a result of new methods used to handle errors occurring across different threads. The following properties that were originally PROTECTED are now PRIVATE and can only be modified through the appropriate methods. Please refer to each properties' documentation for more information.

FieldName CSTRING(MessageMaxlen)
Name of the field that produced the error

FileName CSTRING(MessageMaxlen)
Name of the file that produced the error

KeyName CSTRING(MessageMaxlen)
Name of the key that produced the error

MessageText CSTRING(MessageMaxlen)
Message text

The above four properties have been moved to a PRIVATE ErrorStatusGroup typed GROUP.

Errors &ErrorEntry,PRIVATE
Queue to hold all translated error messages

DefaultCategory ASTRING,PRIVATE
The default category associated with errors that don't have their own category.

Silent BYTE,PRIVATE
Set true to force silent error handling

LogErrors BYTE,PRIVATE
Set true to send errors to the error log

HistoryThreshold LONG,PRIVATE
Set number of items to 'store' in history setting to -1 means keep all errors, setting to 0 switches off history

HistoryViewLevel LONG,PRIVATE
Sets the error level which triggers history viewing, only valid with HistoryThreshold <> 0

HistoryResetOnView BYTE,PRIVATE
Set true to auto clear history after viewing, only valid with HistoryThreshold <> 0

ErrorClass – Methods

GetDefaultCategory (get default error category)

GetDefaultCategory( )

GetDefaultCategory	Retrieves the current default error category.

The **GetDefaultCategory** method retrieves the current default error category set by the DefaultCategory property.

Return Data Type: ASTRING

See Also: SetDefaultCategory

SetDefaultCategory (set default error category)

SetDefaultCategory(*category***)**

SetDefaultCategory	Sets the current default error category.
category	A string constant, variable, EQUATE, or expression that contains the default category.

The **SetCategory** method sets the default error category contained in the property.

See Also: GetDefaultCategory

SetSilent (set silent error flag)

SetSilent(*flag* **)**

SetSilent	Specifies the state of error display mode.
flag	An integer constant, variable, EQUATE, or expression that sets the status of the Silent property.

SetSilent sets the value of the ErrorClass **Silent** private property.

The **Silent** property determines whether an error will be displayed to the screen. If Silent is set to one (1 or True), the error message box will not be displayed to the screen; however it will be added to the error log file. If Silent is set to zero, (0 or False) the error is displayed to the screen as well as added to the error log file.

Example:

```
GlobalErrors.Msg PROCEDURE(STRING Txt,<STRING Caption>,<STRING Icon>,|
LONG Buttons = Button:Ok,LONG DefaultButton = 0,LONG Style = 0)

ReturnValue          LONG,AUTO

  CODE
  SELF.SetLogErrors(TRUE)   !Turn on Error Logging
  SELF.SetSilent(TRUE)      !Set Error Logging to Silent Mode
  ReturnValue = PARENT.Msg(Txt,Caption,Icon,Buttons,DefaultButton,Style)
  RETURN ReturnValue
```

See Also: GetSilent## GetSilent (get silent error flag)

GetSilent( )

GetSilent	Retrieves the current state of error display mode.

GetSilent retrieves the value of the ErrorClass **Silent** private property.

The **Silent** property determines whether an error will be displayed to the screen. If Silent is set to one (1 or True), the error message box will not be displayed to the screen; however it will be added to the error log file. If Silent is set to zero, (0 or False) the error is displayed to the screen as well as added to the error log file.

Return Data Type: BYTE

See Also: SetSilent

SetLogErrors (set error log mode)

SetLogErrors(*flag* **)**

SetLogErrors Specifies the mode of error log activity.

flag An integer constant, variable, EQUATE, or expression that sets the current status of error logging.

SetLogErrors sets the value of the ErrorClass **LogErrors** private property.

The *flag* value turns the error history logging on or off. Setting this value to one (1 or True) turns on the error logging. Setting this value to zero (0 or False) turns off the error logging.

A file with the name “ABCErrors.Log” will be generated in the program folder.

Example:

```
GlobalErrors.Msg PROCEDURE(STRING Txt,<STRING Caption>,<STRING Icon>,|
LONG Buttons = Button:Ok,LONG DefaultButton = 0,LONG Style = 0)

ReturnValue          LONG,AUTO

  CODE
   SELF.SetLogErrors(TRUE)   !Turn on Error Logging
   SELF.SetSilent(TRUE)
  ReturnValue = PARENT.Msg(Txt,Caption,Icon,Buttons,DefaultButton,Style)
  RETURN ReturnValue
```

See Also: GetLogErrors

GetLogErrors (get state of error log)

GetLogErrors( )

GetLogErrors Retrieves the current mode of error log activity.

GetLogErrors retrieves the value of the ErrorClass **LogErrors** private property.

A value of one (1 or True) means that error logging is active. A value of zero (0 or False) means that error logging is inactive (off).

Return Data Type: BYTE

See Also: SetLogErrors, LogErrors

SetHistoryThreshold (set size of error history)

SetHistoryThreshold (*number* **)**

SetHistoryThreshold Specifies the amount of error history items to store.

number An integer constant, variable, EQUATE, or expression that sets the number of items to store in the error log file.

SetHistoryThreshold sets the value of the ErrorClass private property, which sets the number of items to store in the error log file.

Setting the *number* to -1 keeps all errors. Setting *number* to 0 switches off error history logging.

See Also: GetHistoryThreshold

GetHistoryThreshold (get size of error history)

GetHistoryThreshold( )

GetHistoryThreshold Retrieves the current mode of error log history.

GetHistoryThreshold retrieves the value of the of the ErrorClass private property, which sets the number of items to store in the error log file.

A value of -1 keeps all errors. A value of 0 means the error history logging is currently off.

Return Data Type: LONG

See Also: SetHistoryThreshold

SetHistoryViewLevel (set error history viewing mode)

SetHistoryViewLevel(*errorlevel* **)**

SetHistoryViewLevel Specifies the error level to trigger error history.

errorlevel An integer constant, variable, EQUATE, or expression that sets the current error level of error history.

SetHistoryViewLevel sets the value of the ErrorClass private property.

The *errorlevel* value sets the error level for viewing error history. The *errorlevel* value is only valid if the property is set to any value other than 0. Valid errorlevels are as follows:

Level:Benign	no action, returns Level:Benign
Level:User	displays message, returns Level:Benign or Level:Cancel
Level:Notify	displays message, returns Level:Benign
Level:Fatal	displays message, halts the program
Level:Program	treated as Level:Fatal
Level:Cancel	used to confirm no action taken by User

Activating this property will pop up a list box of error messages when the designated error level is posted.

See Also: GetHistoryViewLevel

GetHistoryViewLevel (get error history viewing mode)

GetHistoryViewLevel ( )

GetHistoryViewLevel Returns the active error level set for error history viewing.

GetHistoryViewLevel returns the value of the ErrorClass **HistoryViewLevel** private property.

This value is used to set the error level that will trigger error history viewing. This value is only valid if the HistoryThreshold property is set to any value other than 0.

Return Data Type: LONG

See Also: SetHistoryViewLevel

SetHistoryResetOnView (set error reset mode)

SetHistoryResetOnView(*flag* **)**

SetHistoryResetOnView	Specifies if the error history view structure is cleared on view.
flag	An integer constant, variable, EQUATE, or expression that sets the current reset status of the error history view.

SetHistoryResetOnView sets the value of the ErrorClass **HistoryResetOnView** private property.

The *flag* value determines if the error history view structure should be cleared upon viewing an error message. If *flag* is set to one (1 or True), the History structure will be reset after each error is viewed. Setting *flag* to zero (0 or False) will cause the errors to be queued in the History structure.

See Also: GetHistoryResetOnView, HistoryResetOnView

GetHistoryResetOnView (get the error reset mode)

GetHistoryResetOnView ( )

GetHistoryResetOnView	Retrieves the current status of clearing the error history on view.

GetHistoryResetOnView retrieves the value of the ErrorClass **HistoryResetOnView** private property.

If the value returned is one (1 or True), this indicates that the History structure will be reset after each error is viewed. If the value returned is zero (0 or False) this indicates that the errors are queued in the History structure after viewing.

Return Data Type: BYTE

See Also: SetHistoryResetOnView, HistoryResetOnView

GetFileName (get file that produced the error)

GetFileName( )

GetFileName	Returns the name of the file that produced the error.

GetFileName returns the name of the file that produced the error. The SetFile and ThrowFile methods both set the value of the ErrorStatusGroup **FileName** private property. The **FileName** value then replaces any %File symbols within the error message text.

Return Data Type: STRING

See Also: SetFileName, FileName

SetFileName (set the file that produced the error)

SetFileName(*filename* **)**

SetFileName	Sets the name of the file that replaces any %File symbols within the active error message text.
filename	A STRING constant, variable, EQUATE, or expression that sets the file name to use in the current error message text.

SetFileName sets the value of the ErrorStatusGroup **FileName** private property.

The *filename* value replaces any %File symbols within the current error message text.

See Also: GetFileName, FileName

GetFieldName (get field that produced the error)

GetFieldName( )

GetFieldName Returns the name of the field that produced the error.

GetFieldName returns the name of the file that produced the error. The SetField method sets the value of the ErrorStatusGroup **FieldName** private property. The **FieldName** value then replaces any %Field symbols within the error message text.

Return Data Type: STRING

See Also: SetFieldName, FieldName

SetFieldName (set field name that produced the error)

SetFieldName(*fieldname* **)**

SetFieldName Sets the name of the field that replaces any %Field symbols within the active error message text.

fieldname A STRING constant, variable, EQUATE, or expression that sets the field name to use in the current error message text.

SetFieldName sets the value of the ErrorStatusGroup **FieldName** private property.

The *fieldname* value replaces any %Field symbols within the current error message text.

See Also: GetFieldName , FieldName

GetKeyName (get key name that produced the error)

GetKeyName( )

GetKeyName	Returns the name of the key that produced the error.

GetKeyName returns the name of the key that produced the error. The SetKey method sets the value of the ErrorStatusGroup **KeyName** private property. The **KeyName** value then replaces any %Key symbols within the error message text.

Return Data Type: STRING

See Also: SetKeyName, KeyName

SetKeyName (set the key name that produced the error)

SetKeyName(*keyname* **)**

SetKeyName	Sets the name of the key that replaces any %key symbols within the active error message text.
keyname	A STRING constant, variable, EQUATE, or expression that sets the key name to use in the current error message text.

SetKeyName sets the value of the ErrorStatusGroup **KeyName** private property.

The *keyname* value replaces any %Key symbols within the current error message text.

See Also: GetKeyName, KeyName

GetMessageText (get current error message text)

GetMessageText( )

GetMessageText	Returns the message text of the current active error.

GetMessageText returns the text to substitute for any %Message symbols within the error message text. This value then replaces any %Message symbols within the error message text.

Return Data Type: STRING

See Also: SetMessageText

SetMessageText (set the current error message text)

SetMessageText (*message* **)**

SetMessageText	Sets the message text that replaces any %Message symbols within the active error message text.
message	A STRING constant, variable, EQUATE, or expression that sets the message text to use in the current error message text.

SetMessageText sets the value of the ErrorStatusClass **MessageText** private property.

The *message* value replaces any %Message symbols within the current error message text. The ThrowMessage method sets the value of the MessageText property. The MessageText value then replaces any %Message symbols within the error message text.

See Also: GetMessageText, MessageText

Language Reference Manual

MEMO/BLOB Property handling

MEMO and BLOB fields are now both numbered negatively. MEMO and BLOB declarations begin with -1 and decrement by 1 for each subsequent MEMO and BLOB, in the order in which they appear within the FILE structure.

This affects the following FILE properties (See the Language Reference PDF or online help for specific detail):

PROP:Text

PROP:Value

PROP:Memos

PROP:Type

Variable Size Declarations

Change in Example Code. The correct way to implement variable size declarations:

```
    PROGRAM

    MAP
    TestProc(LONG)
    .

   CODE
   TestProc(200) !can also be an initialized variable


TestProc    procedure(long varlength)

VarString   STRING(VarLength)

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

Also in the example regarding Variable Size Arrays, add the DATA compiler directive:

```
VariableArray  ROUTINE
  DATA
Element     LONG(100)                  !assign number of elements
DynArray    CSTRING(100),DIM(Element)  !declare a variable length array

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

DERIVED prototype attribute

Change in example code. Replaced with the following:

```
ClassA CLASS
Method1 PROCEDURE(LONG, <LONG>),VIRTUAL
       END

ClassB CLASS(ClassA)
Method1 PROCEDURE(LONG,<LONG>),DERIVED
       END

ClassC CLASS(ClassA)
Method1 PROCEDURE(LONG,<LONG>),VIRTUAL,DERIVED
       END

ClassD CLASS(ClassA)
Method1 PROCEDURE(STRING),DERIVED !Will produce compiler error:
       END                        !must match parent prototype
```

ANY

Added to ANY documentation:

Use of ANY in CLASS definitions.

If you do a reference assignment to an ANY that is a member of a CLASS, you must clear the reference *before* destroying the class, otherwise the memory allocated by the reference assignment will not be freed.

Example:

```
AClass CLASS
AnyVar   ANY,PRIVATE
AMethod  PROCEDURE(FILE f)
Destruct PROCEDURE()
       END

TextFile FILE,DRIVER('ASCII')
          RECORD
Line       STRING(255)
          END
         END

  CODE
    AClass.AMethod(TextFile)

AClass.AMethod PROCEDURE(FILE TextFile)
AGroup &GROUP
  CODE
    AGroup &= TextFile{PROP:Record}
    SELF.AnyVar &= WHAT(AGroup, 1)

AClass.Destruct PROCEDURE()
  CODE
    SELF.AnyVar &= NULL  !Without this line the program will leak memory
```

NULL. SETNULL, SETNONULL

NULL, SETNULL and SETNONULL now come in two formats. You can now pass a file as the first parameter to these functions. This allows you to pass references to fields.

Example:

```
SwapNullState PROCEDURE(File F, ANY var)
  CODE
    IF NULL(f, var)
      SETNONULL(f, var)
    ELSE
      SETNULL(f, var)
    END
```

WHAT

WHAT now supports an optional third parameter as follows:

WHAT(*group, number* [*,dimension*] **)**

WHAT	Returns a specified field from a *group* structure.
group	The label of a GROUP, RECORD, CLASS, or QUEUE declaration.
number	An integer expression specifying the ordinal position of a field in the *group*.
dimension	An optional dimension element number, if applicable.

The **WHAT** statement returns the *number* specified field from a *group* structure. Generally, this would be assigned to an ANY variable. If the *number* specified field is a dimensioned field, then **WHAT** returns a reference to the *dimension* element of the *number* field.

If field with an ordinal position is equal to the passed second parameter, and is defined as an ANY type or has the DIM attribute, the returned value can be used only in "reference equality" (&=) operations. Any attempt to access these field types by returned reference will cause a run-time error.

FILEERROR and FILEERRORCODE

When an ERRORCODE 47 (“Invalid Record Declaration”) is returned by a given file operation, you can now use FILEERRORCODE and FILEERROR to get extended information.

FILEERRORCODE and FILEERROR are now valid for ERRORCODE 90 and 47

GETGROUP (return reference to GROUP)

GETGROUP(*group, number* [,*dimension*] **)**

GETGROUP	Returns a reference to a specified *group* structure.
group	The label of a GROUP or QUEUE declaration.
number	An integer expression specifying the ordinal position of a GROUP or QUEUE in the specified *group*.
dimension	An optional dimension element number, if applicable.

The **GETGROUP** statement returns a GROUP reference to the *number* specified field from a target *group* structure. If the *number* specified field is a dimensioned field, then **GETGROUP** returns a reference to the *dimension* element of the *number* field. GETGROUP returns a NULL reference if the *number* specified is not a GROUP. This function is useful if you need to access a field that is part of a dimensioned GROUP.

The returned value can be used only in "reference equality" (&=) operations. Any attempt to access these field types by returned reference will cause a run-time error.

Return Data Type: ANY

Example:

```
MyGroup  GROUP
SubGroup  GROUP,DIM(3)
number     LONG,DIM(5)
          END
         END

gr &GROUP
lr ANY

 CODE
  gr &= GETGROUP(MyGroup, 1, 2)
  lr &= WHAT(gr, 1, 4)
  !lr now references MyGroup.SubGroup[2].number[4]
```

See also: WHAT, ISGROUP, HOWMANY

HOWMANY (return dimensions)

HOWMANY(*label* ,*element***)**

HOWMANY	Returns the number of dimensions for a designated element.
label	The label of a GROUP or QUEUE.
element	A LONG constant or variable that identifies the ordinal position of the target element to examine.

HOWMANY returns the number of dimensions the n'th field of a GROUP or QUEUE. A multi-dimensioned field has its dimensions flattened.

Return Data Type: LONG

Example:

```
MyGroup  GROUP
var1      LONG,DIM(3,3)
var2      LONG,DIM(100)
var3      LONG,DIM(2,3,4)
         END

 CODE
  HOWMANY(MyGroup, 1)  !Returns 9
  HOWMANY(MyGroup, 1)  !Returns 100
  HOWMANY(MyGroup, 1)  !Returns 24
```

See Also: ISGROUP, GETGROUP, WHAT

ISGROUP (return GROUP type or not)

ISGROUP(*label, element* **)**

ISGROUP	Returns true if the *element* is a GROUP data type.
label	The label of a GROUP or QUEUE.
element	An integer expression specifying the ordinal position of a field in the GROUP or QUEUE.

The **ISGROUP** statement returns true if the *element* of a GROUP or QUEUE is a GROUP data type.

Return Data Type: SIGNED

Example:

```
MyGroup  GROUP
F1        LONG         !Field number 1
F2        SHORT        !Field number 2
F3        STRING(30)   !Field number 3
InGroup   GROUP        !Field number 4
F1         LONG        !Field number 5
F2         SHORT       !Field number 6
F3         STRING(30)  !Field number 7
          END
         END

Flag  LONG
 CODE
 Flag = ISGROUP(MyGroup,1)  !returns FALSE

 Flag = ISGROUP(MyGroup,4)  !returns TRUE
```

See Also:

WHAT, WHERE

PROP:CustomColor

A read/write SYSTEM property that can be used to save custom colors added by a user between program sessions.

Syntax:

```
color = SYSTEM {PROP:CustomColor, n}
SYSTEM {PROP:CustomColor, n} = color
```

n must be an integer value between 1-16.

This property returns the RGB value of *n*-th entry of the Custom Colors list contained in the standard Color dialog. If the *n*-th entry is not set, the returned value is 0FFFFFFh (white color).

Setting this property sets *n*-th entry of the Custom Colors list of the standard Color dialog to specified RGB value.

Database Drivers

ADO

The general description of ADO used in Clarion has been amplified as follows:

The Clarion ADO classes and templates were created to provide access to the MS ADO database connectivity layer. While not specifically a database driver it is accessed in a similar manner. As such intrinsic data file operations will not function and require that the provided class methods and templates be used.

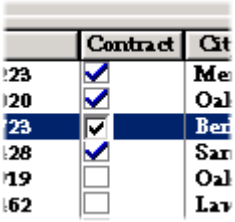
Any reference to “ADO Driver” has been modified to read “ADO interface”

FAQ

How to Update an Icon Check Box in EIP Mode

Introduction:

The release of Clarion 6 introduced a new Edit-in-Place interface which allows better control of column types in a list box. One of those types is a check box interface, implemented through the EditCheckClass object. Instead of a standard entry field that is used in EIP mode, a check box control is used in its place.



Edit-In-Place using EditCheckClass template interface

Problem:

During the edit cycle on each list box line, a check box icon will not be updated until the record (or row) has been saved. This is confusing to a user who unchecks the Edit-In-place check box control, moves to the next column, and does not see the icon check box change state.

Goal of this FAQ:

To update the icon checkbox used while editing, and before the record (row) is saved.

Solution:

The embedded source code added below is used to change the icon on a browse using the values in the associated queue, which have not been updated yet to the primary table (in this example, a check box).

The solution is to add a few lines of source code to the *BRW1::EIPManager.TakeAccepted* method just after the parent call. The same code used to change the icon for each column is used, but we need to reference the browse queue elements. For example,

```
IF (fieldname = 1)
```

would be changed to:

```
IF (BRW1.Q.fieldname = 1)
```

where *fieldname* is a value that controls the display of an icon type, and *BRW1.Q* is the instance prefix of the Browse Box control template

Also the queue fields needs to be changed to reference the parent browse object(SELF to BRWx)

```
SELF.Q.fieldname_Icon = 1
```

to

```
BRW1.Q.fieldname_Icon = 1
```

Icon files used in a browse box are set in the *BRW1.SetQueueRecord* method.

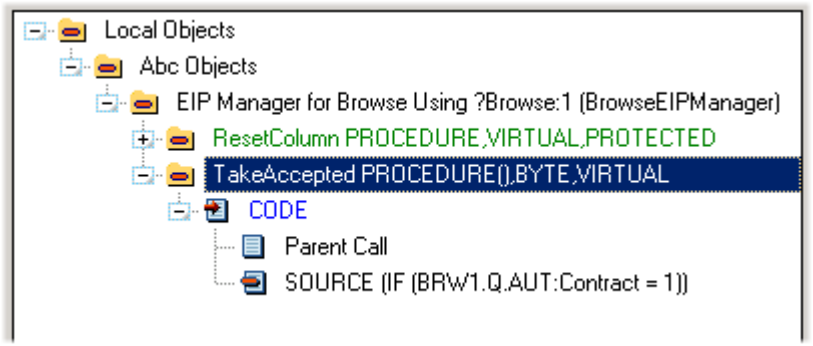
So, the following code:

```
IF (fieldname = 1)
  SELF.Q.fieldname_Icon = 1    ! Set icon from icon list
ELSE
  SELF.Q.fieldname_Icon = 2    ! Set icon from icon list
END
```

should be copied to *BRW1::EIPManager.TakeAccepted* after parent call embed point as follows:

```
IF (BRW1.Q.fieldname = 1)
   BRW1.Q.fieldname_Icon = 1   ! Set icon from icon list
ELSE
   BRW1.Q.fieldname_Icon = 2   ! Set icon from icon list
END
```

The following image shows a real world implementation:



If you are using the Clarion template chain instead of ABC, the embed point and solution is exactly the same, with a minor change to the naming conventions:

```
IF (Queue:Browse:1.BRW1::fieldname = 1)
```

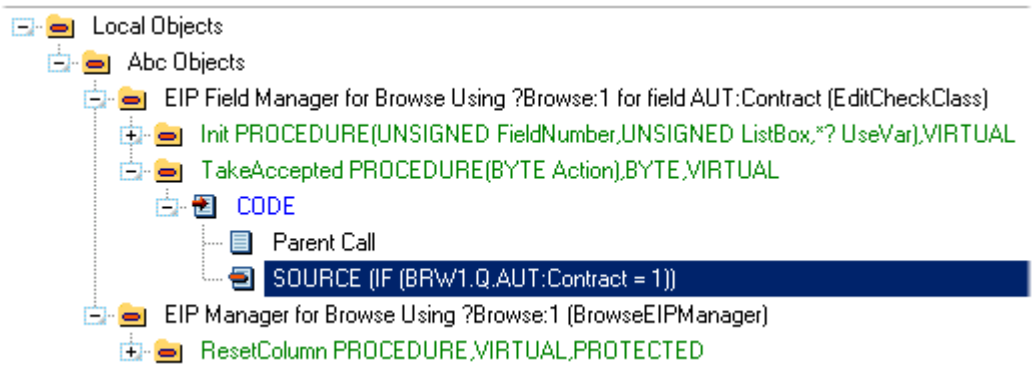
```
 Queue:Browse:1.BRW1::fieldname:Icon = 1       ! Set icon from icon list
ELSE
 Queue:Browse:1.BRW1::fieldname:Icon = 2       ! Set icon from icon list
END
```

Finally, if you are still using the Original EIP implementation (which was only available in the ABC templates), the technique is still the same, but the embed point changes:

```
IF (BRW1.Q.fieldname = 1)
   BRW1.Q.fieldname_Icon = 1   ! Set icon from icon list
ELSE
   BRW1.Q.fieldname_Icon = 2   ! Set icon from icon list

END
```

This code will get inserted into the following embed as shown:



Template Language Reference

#PRINT

Change example code:

```
#PRINT(%ModuleFile, "Printout ' & %ModuleFile)

to

#PRINT(%ModuleFile,'Printout ' & %ModuleFile)
```

#PROCEDURE

Added documentation for the PARENT attribute as follows:

PARENT	Specifies that the procedure has a parent procedure, and inherits all of the prompts and interfaces of that parent procedure.
name	A string constant naming the parent procedure – Example: *window*
family	The name of the template class where the parent procedure name is defined. Example: *ABC*

Example:

```
#PROCEDURE(Splash,'Splash Window'),WINDOW,REPORT,PARENT(Window(ABC))
```

Built-in Template Procedure

TAILNAME (extract file name from full path)

TAILNAME(*symbol*)

TAILNAME	Extract a filename from a symbol, removing any path element.
symbol	The symbol containing the filename to extract.

The **TAILNAME** function processes a full path and file name and removes the path part of the symbol. The returned string is its name and extension only. **TAILNAME** does not expand short names. This function is only available in template language usage.

Return Data Type: STRING

Example:

```
#!  Generate the default install files  ---
#CREATE(%WiseFile2)
Document Type: WSE
item: Global
  Version=7.0
  File Checksum=646309025
end
#INSERT(%InstFile, FULLNAME(%ProjectTarget), '%MAINDIR%\' &
TAILNAME(%ProjectTarget), %WiseFile2Count)
```

See Also: FULLNAME

New Templates

Code Templates

The following XML Support code templates were added to Clarion 6:

FromXML code template

This code template is used to import an XML file's contents into a designated target structure. It is similar to the Import From XML code template, but provides additional settings for more control as to the input contents.

Select Structure

Press the ellipsis button to select a valid label of a GROUP, FILE, VIEW or QUEUE to hold the contents of the import XML file.

Import file

Press the ellipsis button to select a valid XML source file to import. If you wish to select this file name at runtime, check the **Select import file at runtime** check box. You may also specify a variable here (Example: *!FilenameVariable*).

Mapping file (optional)

Press the ellipsis button to select a valid XML mapping file to use during import. In Clarion terms, mapping refers to associating an XML tag or schema to a database column name. If you wish to select this file at runtime, check the **Select mapping file at runtime** check box. You may also specify a variable here (Example: *!FilenameVariable*).

Root Tag:

Enter a hard coded string value, or specify a variable that will identify the root tag name (using the *!variablename* format) to begin the import from.

Row Tag:

Enter a hard coded string value, or specify a variable that will identify the row tag name (using the *!variablename* format) to begin the import from.

Use Schema on import
There are two ways to describe the elements of an XML document. The default is to use a DTD (Document Type Definition) to define the legal building blocks of an XML document. It defines the document structure with a list of legal elements. The alternative method is to use an XML Schema. Check this box to designate the Schema on import instead of the DTD.

Perform a silent import

Check this box if you wish to suppress error generation during the import process.

Customize Columns

Press this button that displays a list box interface where you can add custom mapping from the XML Tag Name to an existing column in your database.

Column

Select a valid label from the drop list. The elements contained here are reads from the **Select Structure** value.

XML Tag Name

Enter the target tag name to match to the selected column.

Picture

Enter a default picture to use to format the imported XML data.

XML Data Format

Select from *Text*, *Cdata*, or *Base64*. The default is *Text*.

XML Style

Select *Tag-based* or *Attribute-based*. This setting controls the XML formatted output style.

ToXML code template

This code template is used to export contents to an XML file's from the designated data origin.

Data Origin

Press the ellipsis button to select a valid label of a GROUP, FILE, VIEW or QUEUE that holds the XML contents to export to an XML file. Press the "**E**" button to call the Expression Editor. This dialog is used to help you construct syntactically correct expressions to use in the **Data Origin** prompt.

Export file

Press the ellipsis button to select the valid XML source file name to export to. If you wish to select this file at runtime, check the **Select export file at runtime** check box. You may also specify a variable here (Example: *!FilenameVariable*).

Mapping file (optional)

Press the ellipsis button to select a valid XML mapping file to use during the export process. In Clarion terms, mapping refers to associating an XML tag or schema to a database column name. If you wish to select this file at runtime, check the **Select mapping file at runtime** check box. You may also specify a variable here (Example: *!FilenameVariable*).

Root Tag:

Enter a hard coded string value, or specify a variable that will identify the root tag name (using the *!variablename* format) to begin the import from.

Row Tag:

Enter a hard coded string value, or specify a variable that will identify the row tag name (using the *!variablename* format) to begin the import from.

XML Style

Select *Tag-based* or *Attribute-based*. This setting controls the XML formatted output style.

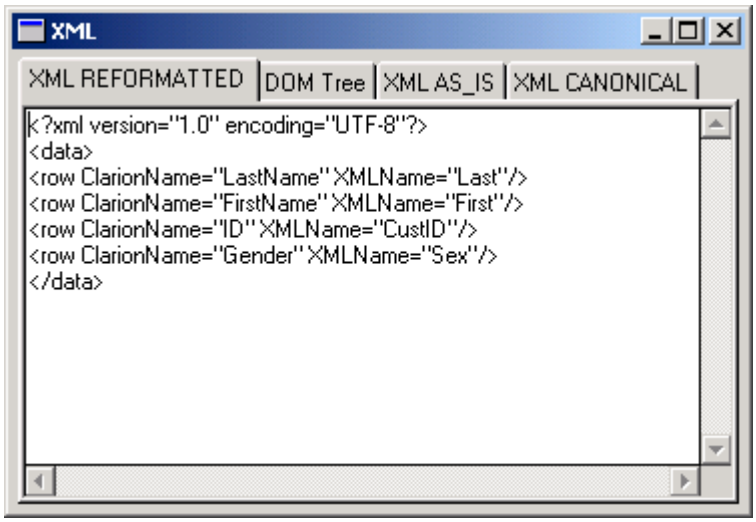
Process Transaction Frame Checkpoint code template

This code template requires the ProcessTransactionFrame.

It is used to generate an even more precise control of the **number of records** that will be included in each transaction frame. You must have the **Generate one transaction for each record read** check box active in the ProcessTransactionFrame extension.

ViewXML code template

The ViewXML Code template is used to analyze an XML file with the use of a default viewer provided by an internal class method. The following window, with four different views, is displayed at runtime:



XML file to view

Press the ellipsis button to select the valid XML source file name to view. If you wish to select this file at runtime, check the **Select file to view at runtime** check box. You may also specify a variable here (Example: *!FilenameVariable*).

Control Templates

New Prompt in table related controls

The following additional prompt has been added to the following control templates (for both Clarion and ABC templates):

File Drop

File Drop Combo

Relation Tree Control

Browse Box

Table Schematic Description

Enter a descriptive string that will be displayed in the Table Schematic window for this particular control. This allows you to distinguish one control from another when there are multiple controls populated in a single window.

Extension Templates

Process Transaction Frame extension template

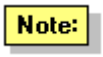
This template is available for any Process procedure in the ABC template family. It is used to override the application's default transaction frame support for any RI operations referenced by the primary table of the process procedure.

The following template prompts are provided:

Generate one transaction for each record read

By default, the Process Transaction Frame extension will enclose a transaction frame around the entire process (Starting the transaction process in the OpenReport method, and finishing the transaction process in the CloseReport method). This gives you an "all or none" default option.

Check this box to allow a transaction frame to be applied to *each* record read. This allows you to create partial updates in the process procedure, in the event that any errors occur on other records processed.



You can even get a greater control on how many records get "framed" with the inclusion of the Process Transaction Frame Checkpoint code template in the embed point of the TakeRecord method. This checkbox must be on for the frame checkpoint to be generated.

Include Children in the transaction?

The primary table named by the process procedure will *always* be included in a transaction frame on any update operation. This option allows you to specify if you want to include any children that are related to the primary table (via the dictionary). Select *Always* if you want to include all child tables in all update operations. Select *Never* if you want to exclude all related child tables from all database operations on the primary table.

List of tables included in the transaction

This list box allows you to include other tables that are not default child tables specifically related to the process procedure's primary table. An example of this may be a "history" table that periodically gets updated under user control. Pressing the **Insert** or **Properties** button displays the following additional prompts:

> **Table**
> Enter the label name of a table that you wish to include in the transaction frame, or press the ellipsis to select a table from the subsequent *Select Table* dialog.
>
> **Include Children in the transaction?**
>
> The options shown here are described in detail in the similar prompt documented above.

Classes Tab

Provides the standard template class interface. By default the supporting TransactionManager object generated by the template is named *Transaction*.

Save Button Transaction Frame extension template

This template is available for any procedure that uses the ABC Save Button control template. It is used to override the application's default transaction frame support for any RI operations referenced in the target procedure.

The following template prompts are provided:

Include Children in the transaction?

The primary table named by the Save Button control template will *always* be included in a transaction frame on any update operation. This option allows you to specify if you want to include any children related to the primary table. Select *Always* if you want to include all child tables in all update operations. Select *Only on Delete and Change* to include the child tables in the transaction frame only during those specific update operations. Select *Never* if you want to exclude all related child tables from all database operations on the primary table.

List of tables included in the transaction

This list box allows you to include other tables that are not default child tables specifically related to the Save Button control template's primary table. An example of this may be a "history" table that periodically gets updated under user control. Pressing the **Insert** or **Properties** button displays the following additional prompts:

Table
Enter the label name of a table that you wish to include in the transaction frame, or press the ellipsis to select a table from the subsequent *Select Table* dialog.

Include Children in the transaction?

The options shown here are described in detail in the prompt documented above.

Classes Tab

Provides the standard template class interface. By default the supporting TransactionManager object generated by the template is named *Transaction*.

Index: