FAQ, Tips and Tricks

SoftVelocity

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

Printed in the United States of America (1003)

Contents:

General Information

The following section contains general information regarding topics, techniques, and programming solutions that are common to most Clarion developers.

Adding Procedure Extensions

Extension and control templates provide additional functionality to basic procedure templates. **Control templates** give your procedure the ability to display and manage specific controls. For example, a browse box may be added using a control template.

Extension templates give your procedure additional functionality not associated with specific controls. For example, date and time displays may be added using an extension template.

From the **Procedure Properties** dialog press the **Extensions** button to display the **Extension and Control Templates** dialog. This dialog displays a list of control and extension templates and the prompts associated with each template. Selecting a template on the left side of the dialog causes the prompts associated with the selected template to be displayed on the right side of the dialog.

Add extension templates by pressing the **Insert** button. Customize existing templates by filling in the prompts on the right side of the dialog.

Only Extension templates may be added and deleted using the Extensions button. Control templates may not be added or deleted, but may be modified. Control templates may be added or deleted from the Window Formatter by adding or deleting their associated controls.

ActiveX Controls, License Files, and Compound Storage Files

ActiveX Controls and License Files

Many ActiveX controls operate on two levels: the development/design level, and a more limited end user level. Typically, a developer buys a license that allows him to operate the control at the design level and to redistribute the control at the end user level. This type of license is often enforced through an associated license file. (.LIC or .LPK). That is, the license file must be present at design time to successfully manipulate the properties of the ActiveX control during application development; but it need not (and should not) be present at runtime to support the more limited end user operation of the control. Thus, the ActiveX control (.OCX and/or .DLL) may be distributed with the application to end users, but the license file need not be distributed, and in fact, cannot legally be distributed under the terms of the typical license agreement.

Clarion for Windows and License Files

Clarion for Windows lets you implement ActiveX controls in your application with the following methods. All the methods require the presence of the license file at design time and the last method requires the presence of the license file at runtime as well; therefore **you cannot use the last method for distribution to end users**.

Method 1:

Within the WINDOW structure, name a compound storage file (.OLR) that contains an instance of the ActiveX control.

Use the OCX Property32 Program to create the compound storage file. Then you can use the Window Formatter to populate an OLE container control (or control template) and name the existing compound storage file in the **Storage File** field in the **OLE Properties** dialog.

Method 2:

Outside the WINDOW structure, name a compound storage file that contains an instance of the ActiveX control.

```
?CalendarObject{PROP:Open} = 'Calendar.OLR\!MeetingSchedule'
```

Method 3 (not for use with distributed applications):

Outside the WINDOW structure, name the ActiveX control. This method requires the presence of a license file at runtime--a violation of the typical license agreement; therefore **you cannot use this method for distribution to end users**.

```
?CalendarObject{PROP:Create} = 'GraphDemoLib.GraphDemo'
```

Benefits of Compound Storage Files

Compound storage files are files that contain one or more OLE or ActiveX objects, including any special property settings for those objects such as fonts, colors, sizes, initial values, etc. Microsoft established the standard format for these files, which in the Clarion for Windows implementation have a file extension of .OLR (by convention only).

You can use compound storage files to reduce and reuse the code needed to implement ActiveX objects in your application. In addition, if you always use compound storage files to implement your ActiveX controls, you will never need to distribute license files to end users in violation of your license agreement.

Compound storage files reduce code. You can use the OCX Property32 Program to create the compound storage file. You can save the custom properties of a specific object by visually manipulating the object (such as a spreadsheet or a calendar control) with its own powerful and easy to use methods, rather than laboriously hand coding property assignments. For example, you can format an Excel spreadsheet using Excel menu commands, toolbar buttons, etc., then save the spreadsheet in a compound storage file, then open the spreadsheet in its current state from the compound storage file. Without the compound storage file, you would have to issue a series of Excel property assignments to the Excel spreadsheet object to initialize it to the desired state.

Compound storage files reuse code because you can reference a single compound storage file many times within a single application or from multiple applications. The property settings saved within the compound storage file are reused with each reference.

Prototyping and Parameter Passing in the Application Generator

When calling procedures and functions, you may want to pass parameters, return values, or both. You can define (prototype) parameters and return values with the **Procedure Properties** dialog.

To pass parameters to a procedure, you must do the following.

1. Add the parameters' data types to the prototype in the program's MAP.

2. Add the parameters' names to the PROCEDURE statement.

3. Pass the parameters in the procedure call.

Adding Parameters to the Prototype

Use the **Prototype** field in the **Procedure Properties** dialog to redefine the Clarion Language prototype statement generated in the program's MAP. The prototype declares everything the calling program needs to know to call the PROCEDURE, including the data types of any parameters.

For example, type *(SHORT,SHORT),SHORT* in the **Prototype** field to generate the following prototype statement in the program's MAP:

```
MAP
...
WindowsControls FUNCTION(SHORT,SHORT),SHORT
...
END
```

Notice the entire text from the **Prototype** field, including the parentheses, is appended to the prototype for the function. The words inside the parentheses are the datatypes of the parameters passed to the procedure or function. The word following the parentheses is the datatype of the value returned by the procedure.

Adding Parameters to the PROCEDURE Statement

Type *(ControlX,ControlY)* in the **Parameters** field in the **Procedure Properties** dialog to generate the following code for the procedure:

```
WindowsControls PROCEDURE(ControlX,ControlY)
  ...
```

Again, the entire text from the **Parameter** field, including the parentheses, is appended to the PROCEDURE statement.

In the **Return Value** field in the **Procedure Properties** dialog, press the ellipsis button (...) to select or define a return variable for the function, and to generate the following code. Note also the generated ProcedureReturn routine now RETURNs the value of the return variable you specified in the **Return Value** field: *ReturnCode*.

```
WindowsControls FUNCTION(ControlX,ControlY)
...
CODE
...
ProcedureReturn ROUTINE
  IF WindowOpened
    CLOSE(window)
  END
  IF LocalResponse
    GlobalResponse = LocalResponse
  ELSE
    GlobalResponse = RequestCancelled
  END
  POPBIND
  RETURN(ReturnCode)
```

You should add embedded code to assign an appropriate value to the returned variable.

Calling the Procedure with Parameters

Use the **Actions** tab of the calling control (menu item, button, etc.) to pass parameters to a procedure or function.

Passing Parameters to Procedures

1. Go to the **Actions** tab for the control.
2. In the **When Pressed** drop-down list, choose *Call a Procedure*.
3. In the **Procedure Name** drop-down list, choose the name of the procedure to call.

   If you have not yet defined the procedure, type it's name. You can define the new procedure later.
4. In the **Parameters** field, type the parameters to pass, separated by commas.

   The parameters may be literal values, expressions, or variable names.

Receiving Return Values from procedures

Although you may call a function with *Call a Procedure* from the **Actions** tab, this method does not allow you to receive return values. Therefore, you should use embedded source to receive a return value from a function. Following is one way to call a function from a control, however, you may call a function in many ways.

1. Go to the **Actions** tab for the control.
2. In the **When Pressed** drop-down list, choose *No Special Action*.
3. Press the **Embeds** button.
4. In the **Embedded Source** dialog, choose the Accepted embed point for *Control Event Handling* then press **Insert**.
5. In the **Select Embed Type** dialog, select SOURCE.
6. In the Text Editor, type your function call, for example:

```
ReturnCode = WindowsControls(0,ControlY)
```

7. Choose **Exit!** from the menu, then save your embedded source when prompted.
8. Set the Priority to 6000

Request and Response

One of the biggest considerations of template design is inter-procedure communication. The added dimension of multi-threading only serves to make this more complex.

In a generic template-driven system, it is impossible to require that parameters be supported in templates. It's never certain if a Browse will be calling a Form, or if it calls a Report, etc. In fact, with Control Templates, a form can also *be* a browse, and an ASCII viewer. To require users to know all of the different parameters and their values is unreasonable. Further, building in support for functions would overcomplicate the templates to the point of not being useful. Again, add in the complications of multithreading and the system is unusable and difficult to maintain.

Using global variables is acceptable, with the THREAD attribute ensuring that the variable itself is safe within a thread. Unfortunately, the value of any global variable must be called into question as soon as any EMBED point is encountered, as the value could change with another procedure call, etc.

Any communications variable must therefore have as little happen from the time it is assigned a value and the time that value is interpreted. This time is referred to as the *Span* of the variable. The shorter span, the better the integrity of the system. The communications variable should also be considered suspect as soon as possible. The amount of time that the variable is considered to have a valid value is referred to as *Live Time*. If a variable has a short *Live Time*, it’s less likely to be subject to misinterpretation, and again system integrity benefits.

To this end, we've implemented a *Request and Response* system in the Clarion templates. This system was created to maintain the integrity of inter-procedure communications in a fully generated system. In other words, if no embedded source is used and no hand-coded modules are used, confidence in system integrity is high.

There are three components to the Request and Response system:

Global Variables: GlobalRequest and GlobalResponse. In GENERATED code, there are no points between the place that either variable is assigned a value and the place that that value is interpreted. These variables are defined as:

```
GlobalRequest       LONG,THREAD

GlobalResponse      LONG,THREAD
```

Note:

If you are creating an application that consists of more than one AppGen created DLL, you MUST check the "Generate Internal Global Data as EXTERNAL" check box for all DLLs except one. Likewise, you MUST check the "Generate Internal Global Data as EXTERNAL" check box for each APP creating an .EXE.

Local Variables: ThisWindow.Request (or SELF.Request), ThisWindow.Response (or SELF.Response), ThisWindow.OriginalRequest (or SELF.OriginalRequest).

ThisWindow.Request (or SELF.Request) and ThisWindow.OriginalRequest (or SELF.OriginalRequest) are assigned value immediately after the procedure begins.

ThisWindow.Response (or SELF.Response) is assigned a value before a bit of code signals the exit of the procedure. Right before the procedure terminates, GlobalResponse is assigned the value of ThisWindow.Response (or SELF.Response).

Enumerated EQUATEs:

These are primarily to increase readability of the code. The actual numbers themselves are inconsequential, with one exception; Request values less than 0 are reserved for use in multi-page systems.

```
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

RequestCompleted    EQUATE (1)          !  Update Completed

RequestCancelled    EQUATE (2)          !  Update Aborted
```

Thread Model FAQ

The following topic provides detailed information regarding thread model terminology and techniques used in Clarion 6

In the new Clarion thread implementation, is there now a requirement for the developer to explicitly lock an object/variable?

Both current and prior Clarion versions do nothing to synchronize access to static data (global non-threaded objects). Therefore the new thread implementation breaks *nothing*. The decision on how to access global non-threaded data is still completely for the programmer to decide.

Practice has shown that there are problems with the automatic swapping schema of threaded data. Therefore in the new thread model data is handled completely differently. The main difference is that in the new implementation every instance of a threaded variable has its own location/address.

Note that without explicit synchronization of active threads, a thread can be suspended by the Windows O/S and another switched to at any time, for example a thread switch could occur in the middle of an assignment such as A = B. The problem that this can highlight comes about if you have more than one thread writing to the same non-threaded variable, which is in itself a questionable coding technique.

How does the locking of an object or variable occur, does it happen automatically whenever it is accessed?

Threaded variables or objects can be accessed without any locking: local data is created on the thread stack and can't be directly accessed (by name) from other threads; access to threaded data is controlled by the RTL via code generated by the compiler. So, possible conflicts can only occur with static data; i.e. _global_ variables declared without the thread attribute.

Now that multiple threads are active concurrently, how can I safely share access to non-threaded static data (global variables) amongst threads?

Just use a synchronization object such as a critical section or mutex. Named mutexes, semaphores, pipes, events are targeted for inter-process synchronization.

What are Critical Sections and what do I need to know how to use them?

Critical Sections allow us to enforce mutual exclusion. Critical section objects provide synchronization similar to that provided by mutex objects, except that critical section objects can be used only by the threads of a single process. Event, mutex, and semaphore objects can also be used in a single-process application, but critical section objects provide a slightly faster, more efficient mechanism for mutual-exclusion synchronization. The important property that critical sections have is that only one thread may have ownership at any one time. If a thread tries to enter a critical section when another thread is already inside the critical section, then it will be suspended, and only resumed when the other thread has left the critical section. This provides us with the required mutual exclusion around a shared resource. More than one thread can be suspended waiting for ownership at one time, so critical sections can be used for synchronization between more than two threads

How can Clarion programs access Critical Sections?

Clarion programs can use a CRITICAL_SECTION (and other synchronization objects) and can do so without diving into their complexity. This can be coded for critical sections as follows:

```
MAP
 MODULE('')
  NewCriticalSection PROCEDURE(),*IcriticalSection
 END
END

Sync &ICriticalSection   !The critical section used to synchronize access to data


CODE
……
Sync &= NewCriticalSection()  ! Create the critical section object

Sync.Wait()                   ! lock the critical section

... code to access shared resource

Sync.Release()                ! unlock the critical section
Sync.Kill()                   ! Delete the critical section
```

What are Mutexes?

Mutexes are another kind of synchronization object. Their goal is the same as for critical sections: provide mutual exclusive access to some shared resource. For example, to prevent two threads from writing to shared memory (global data) at the same time, each thread waits for ownership of a mutex object before executing the code that accesses the shared resource.

There are 2 major differences between critical sections and mutexes:

Critical sections can only be used for synchronization of threads owned by the same process. Mutexes can be used for synchronization of threads that can belong to different processes. If a process creates a mutex with some name and another (or the same) process has created a mutex with that name already, the system does not create a new mutex. It returns another handle to the existing mutex instead and sets error code to 183 (ERROR_ALREADY_EXISTS). Unnamed mutexes are local for the process that created them. Such mutexes can only be used for synchronization of that process's threads.

When we use a critical section for synchronization and the requested critical section is already "occupied", the thread that tries to enter it is suspended until the critical section is available, or the method call that "tries" to enter the critical section can return immediately. The difference is that TryEnterCriticalSection returns immediately, regardless of whether it obtained ownership of the critical section, while EnterCriticalSection blocks until the thread can take ownership of the critical section. With a critical section there is no ability to specify a time limit on the wait. The IMutex.Lock() method behaves in the same manner: the thread that executed this method is suspended until it can gain ownership of the mutex. But it's possible to limit the waiting time by using the IMutex.Wait method with a parameter equal to the number of milliseconds to wait. If the mutex is not released within the given time period, the thread is resumed and the method returns 1. If the mutex is not locked by another thread, or it is released within the given "wait" period, the thread continues its execution and the method returns 0. Supplying a time parameter equal to 0FFFFFFFFh (-1) means an infinite wait.

How does the new thread model impact working with FILE and VIEW structures?

In the new thread model any FILE declared in the procedure or routine local scope is treated as threaded regardless of the presence of the THREAD attribute in its declaration.

VIEWs have no THREAD attribute by syntax, but VIEWs declared in the procedure or routine local scope are treated as threaded. A VIEW declared in the global or module scope is treated as threaded if at least one joined FILE is threaded.

How does the new thread model affect CLASSes ?

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

It follows that in previous versions of Clarion, to handle thread specific information within classes there was a need to have arrays or queues. See for example, Info field of the FileManager class. The only instance of FileManager is responsible for handling all the instances of a FILE. As a result it must retrieve information about current thread context (FileManager.SetThread) on every call to any other method. Now an instance of a threaded class can carry thread related data without involving external structures that require synchronized access to them in the case of preemptive thread switching.

Use of threaded classes can make the whole class hierarchy more complex but Orthogonal. This is because all instances of the same class can share some information. For example, the relation between two tables is independent from any particular class instance: keys, fields and possibly expressions, are the same. So, one class can be split into two or more classes: one carries thread specific information and behavior, and others thread independent data and behavior.

What kind of problems might I run into writing my own code?

You need to remember that access to static variables (a global variable that doesn't have the THREAD attribute) is not synchronized; this means that the value of a variable can be changed at any time from other concurrently running threads. As a result, any condition involving static variables can be changed just after its evaluation, i.e.

```
IF A = 0
! Value of A can be changed at this point from another thread
DoSomething(A)
END
```

The solution to this type of problem is to use the THREAD attribute on declarations, or use synchronization if you actually intended for the variable to be shared across threads.

You also need to know that the error functions (i.e., ERRORCODE( )) are now thread dependant. This means that an error raised on one thread are not detectable on other threads. This may affect some of your code if you are using error states to transmit information between threads.

What has the new thread model changed that I must understand?

In previous Clarion versions all instances of a threaded variable shared the same location in the program. Therefore, the result of a reference assignment with a threaded variable, or the field of a threaded structure as a source is valid for all threads, for example:

```
F FILE,...,THREAD
...
END

A ANY

CODE
A &= WHAT (F.Record, 1)
```

In all threads **A** points to the first field of the file's record. In the new model every instance of a threaded variable has its own address. Therefore, the result of a reference assignment as given above is valid only in the thread where it has been executed.

There are a number of structures in ABC which use a field with thread numbers to distinguish data for a particular thread. Examples are FileThreadQueue or StatusQ in ABFILE.CLW. The fact that every instance of a threaded variable now has its own address required that the schema for initialization of most FILE-related ABC classes were in need of some changes. (We have already done the changes and will utilize same as a teaching tool).

On the other hand, global reference variables set to particular instances of threaded variables can be useful for passing information across threads: for example one thread can change an instance belonging to another thread.

Has the syntax of data declarations changed?

No, no change in syntax. However, because the address of the instance for the starting thread is only known at compile time the compiler must generate code to calculate the address of a required instance. There are two possibilities here: the variable is External and imported from another DLL, or it is located in the same executable.

So, one change is that the EXTERNAL and THREAD attributes in data declarations are not mutually exclusive now. The THREAD attribute is the only way to inform the compiler that it must generate the code to get the correct instance.

How can I reference a threaded variable from another thread?

The new INSTANCE function: INSTANCE(variable,threadno) returns the address for the instance of the variable allocated for the specified thread referenced by threadno, provided the referenced thread is active.

This can be used in code as follows:

```
addressvar = INSTANCE(SalesFile,THREAD())

!return address of SalesFile entity on the active thread
```

INSTANCE is also valuable when you need the thread independent ID of the variable.

```
addressvar = INSTANCE(GLO:LoginID, 0)

!get the thread independent ID of a global threaded var
```

Is the RTL now re-entrant?

The Clarion RTL has per-process and per-thread initialization flags in the header. So, every process will get its own copy of the RTL and Windows does not load the same DLL into process memory space a second time (unless it has been unloaded manually). So Yes, it's possible to consider the RTL as re-entrant.

Using Wizard Options

Wizard Options in the Data Dictionary Editor provide more control over the wizards' functionality. Wizards use the Options specified for a file, field, key, or alias when creating procedures. In addition, the Wizards use file, field, key, and alias names and descriptions for the text on menus, title bars, tabs, etc.

File Options

Do Not Auto-Populate This File

Directs the wizards to skip this file when creating primary Browse procedures or Report procedures.

User Options

User Options let you provide information to utility templates. User Options are comma delimited, that is, a comma separates each entry. Choose from the following:

EDITINPLACE The Browse Wizard provides edit-in-place updates to the browsed file instead of a separate update (form) procedure. We recommend this option for files with one-way lookup relationships, such as a State Code file. Files with complex relationships are better managed with a separate update procedure.

Alias Options

Do Not Auto-Populate This Aliased File

Directs the wizards to skip the Aliased File when creating primary Browse procedures or Report procedures.

User Options

User Options let you provide information to utility templates. User Options are comma delimited, that is, a comma separates each entry.

Column (Field) Options

Do Not Auto-Populate This Field

Directs the wizards to skip this field when creating Form, Browse or Report procedures.

Population Order

Specifies the order in which the wizards populate fields. Choose Normal, First, or Last from the drop-down list. Wizards populate in this order: all Fields specified as First, then all Fields specified as Normal, and finally all Fields specified as Last.

Form Tab

Specifies the TAB onto which the wizards populate the field. Type the Caption for the TAB or select one you have previously created from the drop-down list. This lets you direct the wizard to group fields in the manner you want.

Add Extra Vertical Space Before Field Controls on Forms

Check this box to direct the wizards to add vertical space between this field's control and the one populated above it.

User Options

User Options let you provide information to utility templates. User Options are comma delimited, that is, a comma separates each entry.

Key Options

Do Not Auto-Populate This Key

Directs the wizards to skip this Key when creating primary Browse procedures or Report procedures.

Population Order

Specifies the order in which the wizards populate keys. Choose Normal, First, or Last from the drop-down list. Wizards populate in this order: all Keys specified as First, then all Keys specified as Normal, and finally all Keys specified as Last.

User Options

User Options let you provide information to utility templates. User Options are comma delimited, that is, a comma separates each entry.

Relation Options

User Options

User Options let you provide information to utility templates. User Options are comma delimited, that is, a comma separates each entry.

How to…

How to Add Control Templates

When starting with a new procedure, to add a Control template:

1. In the Window Formatter or Report Formatter, add a Control template by clicking on the tool in the Controls toolbox.
2. Choose a Control template from the **Select Control template** dialog, then place the control on the window or report by clicking on the desired location.

   The formatter places one or more controls (the type of controls depend on the Control template) in the window or report.
3. RIGHT-CLICK on the control, then choose **Actions** from the popup menu to access the Control template prompts.

   These prompts define and customize its functionality.
4. Select the other tabs on the **Properties** dialog to set the control's appearance, location, and other functionality.

Once a Control template is added to a procedure, a check box appears next to the **Extensions** button in the **Procedure Properties** dialog. You can access the Control template prompt with this button.

How to Add a Reverse Sort Order to a Browse

To implement a reverse sort order to a template-generated list box is a simple process of populating a tab, and setting the conditional behavior using the Additional Sort order option.

Set up a new tab from the Window Formatter:

1. Press the Tab Control button on the toolbox.
2. Click on the Sheet control.
3. Change the text to "Reverse Order" or the text you want to display on the tab.

Set up the list box to work with the new tab:

1. RIGHT CLICK on the list control and choose Actions from the popup menu.
2. Select the Conditional Behavior tab.
3. Press the Insert button.
4. Type *CHOICE(?CurrentTab) = n* (where n = the number of the Reverse Order tab) in the condition entry field.
5. Type *-FIL:FieldName* (where FIL:FieldName is the label of the reverse sorted field) in the additional sort field entry field. (Notice the minus (-) sign before the field name)

   Note: For a case insensitive sort, use *-UPPER(FIL:Fieldname)*
6. Press the **OK** button twice, and exit the Window Formatter.

This sets the new tab to sort the list in the reverse order of the field specified.

How to Add a Toolbar

You may add a toolbar to any window with a simple command in the **Window Formatter**: choose the **Toolbar ▸ New Toolbar**. You may place any control on a toolbar, but the ones you will probably use the most are command buttons, check boxes, radio buttons, and drop down list boxes. As with menus, Clarion will automatically merge toolbars in certain situations.

How to Add a Toolbar Command Button

The following describes how to add a toolbar with a command button to a window. The starting point is the **Window Formatter**, open to an empty window:

1. From the **Toolbar** menu, choose **New Toolbar**.

A rectangular area appears at the top of the window. This is the toolbar. At runtime, it appears dark gray.

2. Optionally choose the **Options ▸ Grid Settings,** then check the **Snap to Grid** box.

This makes sizing and placing the controls easier.

3. Select the **Button** icon (**OK**) in the Controls toolbox, then CLICK inside the new toolbar in the sample window.

A button control appears.

4. RIGHT-CLICK on the button and select **Properties** from the popup menu, or choose **Edit ▸ Properties.**

The **Button Properties** dialog for the new button appears.

5. Delete the default text in the **Text** field.

This allows you to create a picture button without text.

6. Type a descriptive Field Equate Label in the **Use** field.

For a File/Open button, for example, you might type ?OpenButton. The Field Equate Label will appear in the **Embedded Source** dialog, making it easy to identify where to embed source.

7. From the **Extra** tab, choose an icon from the **Icon** drop down list, or type the name of an icon file (*.ICO) of your own.

The icon list contains a number of default icons for such standard actions as File/Open, or Cut, Copy, and Paste.

8. Add functionality to the button.

Select an STD ID from the drop down list, or select the **Actions** tab and embed source code, call a procedure or run a program.

9. Press the **OK** button to close the **Button Properties** dialog.

10. Resize the button to the size you want by dragging its handles.

Clarion's . ICO files are 32 x 32 pixels . Most toolbar buttons will be smaller--for example, 16 x 18 pixels. By using these larger files, we can create the "disabled" icon from the same file, rather than requiring a separate file. When creating a custom .ICO file for a toolbar button, place the image in the center of the icon file. Clarion automatically crops the image to fit the button size.

How to Add a "Latched" Button

A latched button "stays depressed" when CLICKED, then returns to its original state when CLICKED a second time. To place latched button:

1. Select the **Check Box** icon in the Controls toolbox, then CLICK inside the new toolbar in the sample window.

The **Select Field** dialog appears.

2. Highlight Local Data, then press the **Insert** button.

The **New Field Properties** dialog appears.

3. In the **Field Name** field, type a name, then choose **BYTE** from the data type drop down list.

The **Check Box Properties** dialog appears. A button created from a check box control has two modes: on or off. When the check box is 'on' (the button appears 'pushed in'), and the value of its USE variable is one. When the check box is 'off' (the button appears raised), and the value of its USE variable is zero.

4. From the **Extra** tab, choose an icon from the **Icon** drop down list, or type the name of an icon file (*.ICO) of your own.

This is what makes it a "latched button" - placing an ICON on a CHECK control.

5. Press the **OK** button.

The button is complete; you need only adjust its position by dragging its center, if necessary.

How to Add a Toolbar Button Group

A button group provides the user with **mutually exclusive choices**. For example, in a group of three buttons, only one can be "depressed." If button number two is currently "depressed," push in button number one, and button number two pops out. A button group can provide controls for left, right and center text justification--only one option can be active at a time.

To create a button group:

1. CLICK on the Option Box icon in the Controls toolbox, then CLICK inside the toolbar.

The **Window Formatter** places an Option Box on the toolbar. You may resize it by dragging its handles. An Option Box--an OPTION structure--must always surround radio button choices, however, this Option Box will not appear on the toolbar, because you will hide it.

2. RIGHT-CLICK on the Option Box and choose **Properties** from the popup menu.

The **Option Properties** dialog appears.

3. Press the ellipsis (...) button for the **Use** field, and define a string variable.

The variable may be global, module, or local data, or it may be a data dictionary field. The variable will receive the **Value** text from the button selected by the user. If you don't specify any **Value** text, it gets the **Parameter** text from the selected button. If you define a numeric variable, it will receive an integer value corresponding to the selected button, that is, button 1, 2, or 3**.**

4. From the **Extra** tab, uncheck the **Boxed** box.

This hides the Option Box from the user. It appears in the **Window Formatter** dialog, but will not appear at runtime.

5. Press the **OK** button.

6. CLICK on the Radio Button icon in the Controls toolbox, then CLICK inside the Option Box.

The Application Generator places a Radio Button where you clicked in the Option Box.

7. RIGHT-CLICK on the Radio Button and choose **Properties** from the popup menu.

The **Radio Button Properties** dialog appears.

8. Clear the **Text** field.

Clearing this field will remove text from the button so we can add an icon with no text.

9. In the **Value** field, type "Left."

When the user presses this button, the string "Left" is assigned to the USE variable we specified above.

10. From the **Extra** tab, choose an icon from the **Icon** drop down list, or type the name of an icon file (*.ICO) of your own.

Adding an icon causes the radio button to look like a command button.

11. Press the **OK** button.

The first button is complete; you need only adjust its position by dragging its center.

12. Repeat steps 6 through 11 for the "center" and "right" buttons.

13. Choose **Preview!** from the **Window Formatter** menu.

This displays the window, including the toolbar and menus, as it would to the user at runtime. Test the latching and radio features by pushing the buttons. Press ESC when done previewing your window.

14. Choose **Exit!** from the **Window Formatter** menu to save your window.

How to Merge a Toolbar

Global and Local Tools

The TOOLBAR structure declares the tools displayed for a window. On an APPLICATION window, the TOOLBAR defines Global tools available to all the windows in the application. However, if the NOMERGE attribute is specified on the APPLICATION's TOOLBAR, the tools are local and are displayed only when no MDI child windows are open; there are no global tools. Global tools are active and available on all MDI child windows unless an MDI child window's TOOLBAR structure has the NOMERGE attribute.

MDI Windows

On an MDI child window, the TOOLBAR defines local tools that are automatically merged onto the Global toolbar. Both the Global and the local tools are then active while the MDI "child" window has input focus. Once the window loses focus, its specific tools are removed from the Global toolbar. If the NOMERGE attribute is specified on an MDI child window's TOOLBAR, the local toolbar replaces the Global toolbar.

Non-MDI Windows

On a non-MDI WINDOW, the TOOLBAR is **never** merged with the Global menu. A TOOLBAR on a non-MDI window always appears in the window, not on any parent window which may have been previously opened.

Merging Order

When an MDI window's local TOOLBAR is merged into an application's global TOOLBAR, the global tools appear first, followed by the local tools. The toolbars are merged so that the tools in the local toolbar begin just right of the position specified by the value of the width parameter of the global TOOLBAR's AT attribute. The height of the displayed toolbar is the maximum height of the "tallest" tool, whether global or local. If any part of a control falls below the bottom, the height is increased accordingly.

Note: To merge toolbars, the global toolbar's AT width must be less than the APPLICATION's frame width.

How to Add Drag and Drop to a List Box

Drag and Drop capability for lists means the user can select an item in a list box, hold down the left mouse button, "drag" the item to another control, release the mouse button to "drop" the item on the control, which can look at the data that was "dropped" on it, and then do something with it.

Adding Drag and Drop to a Clarion list box is a simple operation. This section provides an example of dragging an item from one list box to another, within the same application. You can also "Drag and Drop" to or from another application--for example, File Manager.

To implement Drag and Drop, you must add the DRAGID and DROPID attributes to the controls. You can add either or both to a control. The simplest, quickest way to do this is with Property Syntax statements. Assume for this example that the field equate labels for the two list boxes are ?FromList and ?ToList. Assume you want the end user to be able to drag **from** ?FromList **to** ?ToList.

Set up ?FromList as a **drag host**:

1. RIGHT-CLICK on the list control and choose **Properties** from the popup menu.
2. Select the **Extra** tab.
3. In the **Drag ID** field, type "FromList."
4. Press the **OK** button.

This sets the Drag ID signature which identifies "FromList" as the source of any "drag" operation from this control.

Set up ?ToList as a **drop target**:

1. RIGHT-CLICK on the list control and choose **Properties** from the popup menu.
2. Select the **Extra** tab.
3. In the **Drop ID** field, type "FromList."
4. Press the **OK** button.

This sets the Drop ID signature which specifies that the list will accept any "drop" operation with a Drag ID signature of "FromList."

Add drag functionality to the drag host, that is, detect a drag event and provide something to drag and drop:

1. RIGHT-CLICK on the FromList control and choose **Embeds** from the popup menu.

2. Locate the "Control Event Handling--*?Browse:*1--Drag" embed point and embed the following code:

```
IF DRAGID()
! Doesn't matter who dropped it for now
 SETDROPID('string to drag and drop') ! Passing a simple string
END
```

This code detects a drag event--at the time the user **releases** the mouse button over a valid drop target--and places a string to drag with the SETDROPID function.

You can just as easily use the CHOICE() and GET() functions to retrieve an item from the local QUEUE for the first list box, then place the item in a global QUEUE. Then, upon detecting a drop event in the second list box, you could ADD from the global QUEUE to the local QUEUE for the second list box.

Add drop functionality to the drop target, that is, detect a drop event and retrieve whatever was dropped:

1. Set the Priority to "First".

2. RIGHT-CLICK on the ToList control and choose **Embeds** from the popup menu.

3. Locate the "Control Event Handling--*?Browse:1*--Drop embed point and embed the following code:

```
MyField = DROPID()          ! Retrieve the passed string
CallMyProcedure             ! Handle the rest in procedure
```

4. Set the Priority to "First".

This code detects a drop event--at the time the user **releases** the mouse button over the drop target--and retrieves the "dropped" string with the DROPID function.

You can just as easily use the CHOICE() and GET() functions to retrieve an item from the local QUEUE for the first list box, then place the item in a global QUEUE. Then, upon detecting a drop event in the second list box, you could ADD from the global QUEUE to the local QUEUE for the second list box.

How to Add Embedded Source Code

- RIGHT-CLICK the procedure in the Application Tree, then choose **Embeds** from the popup menu (or press the **Embeds** button in the **Procedures Properties** dialog) to open the **Embedded Source** dialog to embed source code using alphabetically or logically ordered named embed points.

  or

- RIGHT-CLICK the procedure in the Application Tree, then choose **Source** from the popup menu to open the Embeditor to embed source code within the context of surrounding generated code.

  or

- RIGHT-CLICK on a control in the Window Formatter, then choose **Embeds** from the popup menu to access the embed points for a single control.

Clarion's templates let you add your own customized code to many predefined points inside the standard code that the templates generate. It's a very efficient way to achieve maximum code reusability and flexibility. The point at which your code is inserted is called an *Embed Point*. Embed points are available at all the standard events for the window, the window controls, and for many other logical positions within the generated code. The embed points are determined by the templates. You can even add your own embed points if needed. See #EMBED.

Embedding source code in a procedure lets you fully customize the procedure. The Application Generator saves the embedded source in the .app file and integrates it into the template generated source code each time you generate source code.

You can write your own embedded source code or use Code templates to generate the code for you. Once you embed source code in a procedure, the procedure is flagged with an "S" in the Application Tree.

In order to effectively embed code, you should understand the surrounding template generated code. See *Learning Clarion* and the *Template Guide* for more information on the Clarion and ABC Templates and the code they generate.

Several ways to Embed Source Code

Clarion provides several powerful methods for embedding source code. There are advantages to each of these methods as noted below:

- The Embeditor (choose **Source** from the popup menu) lets you see the embedded source code *within the context of the surrounding generated code* and gives you the full power of the Text Editor, including text search and replace, copy and paste, the Populate Fields toolbox, and File Import.

- The **Embedded Source** dialog (choose **Embeds** from the popup menu) lets you see *only the embed points and their code, without the surrounding code.* It gives you the full power of the Text Editor, plus a locator to find embed points, plus tools for moving and copying entire embed points with multiple blocks of embedded code, and for generating embedded code with Code templates.
- The **Embeds** button for a control (choose **Embeds** from the Window Formatter's popup menu) gives you the power of the **Embedded Source** dialog focused on *the embed points for a single control*.

Source code embedded with the Embeditor is fully accessible with the **Embedded Source** dialog and vice versa.

Using the Embeditor

1. From the Application Tree, RIGHT-CLICK the procedure, then choose **Source** from the popup menu.

   The Embeditor generates a temporary source file with optional comments and shading to identify all the embed points for the selected procedure. You may insert source code into the embed points simply by typing the new source statements into the unshaded or white area.

You may configure the Embeditor's temporary source file with the Application and Editor tabs of the Application Options dialog. Choose Setup ▸ Application Options.

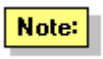
The Embeditor is the Text Editor opened in a special mode which allows you to not only edit all the embed points in your procedure, but to edit them within the context of template-generated code. The Embeditor displays *all possible* embed points for the procedure within the context of *all* the *possible* code that *may* be generated for the procedure. Notice the distinction here--**Embeditor does not show you the code that *will* be generated, but all the code which *could be* generated**, if you placed code into every available embed point.

2. Press to scroll to the next embed point.

   scrolls to the previous embed point; scrolls to the next filled embed point; scrolls to the previous filled embed point.

3. Place the insertion point in the unshaded area, then type your source code.

   The full power of the Text Editor is at your disposal. See *Text Editor* for more information.



The Embeditor automatically indents your source code at least as far as the embed point comments. You may indent farther (to the right), but you may not indent less (to the left).

4. Choose **Exit!** from the menu, then save when prompted.

   The Embeditor automatically puts your source into the appropriate embed point and sets the priority for the embedded code.

Embedded Source Dialog

1. From the Application Tree, RIGHT-CLICK the procedure, then choose **Embeds** from the popup menu.

This opens the **Embedded Source** dialog, providing access to all the embed points in the procedure. You can also get here from the **Embeds** button on the **Procedure Properties** window, but the popup menu is quicker.

You may sort the embed points in alphabetical order or in logical order with the Application tab of the Application Options dialog. Choose Setup ▸ Application Options from the menu.

Filter the embed points list by choosing from all available embeds or only "filled" embeds. Locate an embed point by typing its name in the locator field near the top of the dialog.

You may configure the available list of embed points with the Application tab of the Application Options dialog. Choose Setup ▸ Application Options.

To embed code associated with a specific control, open the Window Formatter, RIGHT-CLICK the control and choose Embeds from the popup menu. Only those embed points associated with the selected control are listed.

2. Select an embed point then press the **Insert** button.

   This opens the **Select Embed Type** dialog. There are three ways to create the embedded source code: hand-coding with the text editor, calling another procedure, or embedding a Code template.

   You may combine one or more of these three methods at a single embed point--that is, a single embed point accepts multiple "blocks" of embedded code. You can control the execution sequence of each block of code relative to any other code in the embed point by setting its priority. Lower priority numbers execute before higher priority numbers.

   The **Embedded Source** dialog displays the embedded source in the order it generates and executes.

To "hand-code" embedded source with the Text Editor

1. Select SOURCE in the **Select Embed Type** dialog.
2. Press the **Select** button to start the Text Editor with a blank source code window.

   This opens the Text Editor. The display includes a Populate Field toolbox from which you can select variable names and field names. Simply CLICK on an item in the toolbox to insert its fully qualified name at the insertion point.
3. Write your custom code in the source code window.

Don't forget to use the on-line help for explanations and examples of Clarion Language syntax and techniques. Copy and paste directly from the help examples!

4. Choose **Exit!**.
5. Choose **Yes** when prompted to save the embedded source.
6. Set the **Priority** for the embedded source.

   The **Priority** of each block within an embed point controls the execution sequence of the code relative to any other code in the same embed point. Lower priority numbers execute before higher priority numbers. Set the custom priority number in the spin box. Standard template generated code generally takes the default priority, so you can place your code before or after the template generated code by setting a lower or higher priority number.

Call a Procedure

1. Select *Call a Procedure* in the **Select Embed Type** dialog.

   A dialog named for the embed point opens to accept the name of the procedure to call.
2. In the **Procedure to Call** field, type a name for the procedure or choose an existing procedure from the drop-down list.

   Typing a new name tells the Application Generator to add the procedure to the Application Tree as a "To Do" item. If another procedure with the same name already exists, the Application Generator generates code to call it.

   You define the functionality of the other procedure separately.
3. Press the **OK** button to close the dialog.

Use a Code template to generate the embedded code

1. In the **Select Embed Type** dialog, select a Code template then press the **Select** button.

   Code templates are the items indented beneath the Class folders.

   This displays a **Prompts for**... dialog box.

2. Read the instructions and explanations in the dialog.

   Each code template includes explanatory text on its proper use and how to fill in the necessary options.

3. Fill in or choose from the options in the **Prompts for**... dialog.

4. Press the **OK** button to close the dialog.

Managing Embedded Source

The **Embedded Source** dialog contains several tools that let you control the sequence in which embedded source is listed and executed. Buttons are available to change the order of multiple embedded source items; execution occurs in the order they are listed.

There are also **Delete** and **Properties** buttons, plus Cut, Copy, and Paste buttons for maintenance. To Cut and Paste (or Copy and Paste) embedded source from one embed point to another:

1. In the **Embedded Source** dialog, highlight a line in the tree diagram.

   Highlighting an embed point line (folder icon) selects *all* the embedded source at this embed point for subsequent cut and paste operations. Highlighting a single embed source item selects only that item.

2. Press the cut or copy button to move an embed point to the clipboard.

3. Again, highlight a line in the tree diagram.

4. Press the paste button to move the embed from the clipboard.

Copying Embedded Source Between Procedures

Occasionally you will create two or more procedures that are very similar and that require lots of embedded source code. Rather than retype the embedded source in the similar procedures, you can copy the embedded source as follows:

1. Develop and test the embedded code in your first procedure.

2. Choose **File ▸ Selective Export**.

3. Specify a .TXA file to receive the exported procedures, then press **OK**.

4. Select all the similar procedures for export, then press **OK**.

Selected procedures are identified with a check mark.

5. With your favorite text editor, open the .TXA file and copy the embed definitions from the finished procedure to the other similar procedures, then save the .TXA file.

   The embed definitions commence with the [EMBED] line. See the *Programmer's Guide* for more information on TXA file format.

6. In Clarion Choose **File ▸ Import Text**.

7. Specify the .TXA file, then press **OK**.

   The import replaces the procedures in the .APP with the procedures from the .TXA, with the embedded source code intact.

How to Add Fields (Columns) to Data Dictionary Files

1. Press the **Fields/Keys** button to open the **Field/Key Definition** dialog.
2. On the **Fields** tab, press the **Insert** button.

   The **New Field Properties** dialog appears.
3. On the **General** tab, type in the field **Name**, choose **Data Type**, specify length in **Characters**.
4. Select the **Validity Checks** tab, and choose a field validation option.
5. Select the **Window** tab to specify how the field and its prompt appear as controls in your application windows and dialogs.
6. Select the **Report** tab to specify how the field will appear on printed reports.

   The specifications on the Window and Report tabs establish defaults for the field. You can always change the settings on a case by case basis.
7. Press **OK** to complete this field and define the next one.

   The **New Field Properties** dialog appears again, ready for the next field.
8. Repeat steps 3 through 7 for additional fields within this file.

   After each field is completed, the **New Field Properties** dialog appears, ready to accept the next field.
9. After adding the last field, press the **Cancel** button in the **New Field Properties** dialog to return to the **Field/Key Definition** dialog.

How to Add Files (Tables) to a Dictionary

1. Press the **Add File** button, then, when asked if you want to use Quick Load, press the **No** button.

   The **New File Properties** dialog appears.

2. On the **General** tab, type the **Name**, the **Prefix**, and choose the **File Driver** for your data file.

3. Press **OK** to close the dialog.

   Your file is added to the dictionary. You may, of course, specify additional file properties if you want. You may add more files by repeating the above steps.

See also:

How to Import a File Definition From an Existing Data File

How to Add a File Alias to the Dictionary

An alias creates a second reference for a file without duplicating the file on disk. You can add an alias for a file only if it's already on the Dictionary list. In the Dictionary dialog, press the **Add Alias** button and fill in the **New File Alias** dialog.

A file alias creates an additional record buffer for a file *on the same thread*. That is, an alias lets you define and use two or more *different* relationships between the *same two files on the same thread*. This is really the only compelling reason to use file aliases, since aliases use additional memory and system resources (file handles).

When using aliases it is best to use a file driver that stores keys internally, such as TopSpeed or Btrieve, to conserve file handles.

For example, let's say your hospital application has a patient file and a doctor file. A patient has several doctors: an admitting doctor, a primary doctor, and a surgeon. In database terms, the patient record has three *different* fields containing doctor IDs, and all three are linking fields to the doctor file (thus, three relationships between the same two files). If you want to automatically display all the patient's doctors on the same window, you need a record buffer for each link, otherwise, you can show only one doctor at a time--the last one retrieved.

By defining aliases for the doctor file, you can supply additional buffers to hold more than one doctor record at a time. Do not confuse this with the THREAD attribute for a file. The THREAD attribute provides for a separate record buffer for each *different* thread, whereas an alias provides an additional record buffer on the *same* thread.

When using aliases, you must open the file for shared access.

You can edit the fields and keys for the Alias by pressing the **Fields/Keys** button. The **Field/Key Definition** dialog lists the fields and keys for the **original** file; any changes you make will update the originals.

How to Add Hot Key Display to a Menu Item

Many programs have "direct action" Hot Keys assigned to commonly used menu items. The usual way to display these is aligned to the right in the item's text. Here's the "secret" to aligning the hot keys -- the ASCII tab character (ASCII 9).

Make your menu text look like this:

```
ITEM('&Customers<9>Alt+F3'),USE(?ViewCustomers),KEY(AltF3)
```

You have to do this through the ellipsis (…) button on the Procedure Properties window, because the Window Formatter will turn the <9> in your text into <<9> (making <9> a constant that appears in the text instead of the ASCII character 9). You can initially enter it into the item's text in the Window Formatter, then go into the ellipsis button and remove the extra leading <. Once you've done it in the code, it'll stay.

How to Assign an Image to Display at Runtime

The parameter for an IMAGE control cannot accept a variable; however, you can reassign the image to display a runtime using a property assignment statement.

Insert the following line of source code in the embed point where the assignment will take place.

?Image{PROP:Text} = FileName

Optionally, you can use the DosFileLookup control template to allow a user to select the graphic image from a standard File Dialog.

How to Auto-size all Columns in a Browse Box when the Window Opens

A new feature available in Clarion 6 is the auto-sizing of Browse Box columns available from the Global Properties **App Settings** tab control. At runtime, double-clicking on a column auto sizes the column to fit the data contents.

There is also a simple way to get all the columns in a browse box to auto size when the window opens.

Set the following property after the browse box is initialized, and the list box has been populated:

```
BRWx::AutoSizeColumn.ResizeAll()
```

where *BRWx* is equal to the instance of the browse object. A good embed point is the *very last embed point* provided by the Window Manager's Init method:

```
! Prepare Alert Keys
SELF.SetAlerts()
! [Priority 9001]

BRW1::AutoSizeColumn.Init()
BRW1::AutoSizeColumn.AddListBox(?Browse:1,Queue:Browse:1)
! [Priority 9550]
BRW1::AutoSizeColumn.ResizeAll()
! End of "WindowManager Method Executable Code Section"

RETURN ReturnValue
```

How to Change the Printer Device without calling PRINTERDIALOG

You can change the windows default printer without calling the PRINTERDIALOG function. You may want to do this when a report is designed for pre-printed forms, and therefore must always be routed to a printer loaded with the forms.

This can be done by using Clarion's property syntax. The property to use is **PROPPRINT:Device**. This property definition can be found in the ..\LIBSRC\PRNPROP.CLW. This must be included in your application before making use of any of the properties defined therein.

To include PRNPROP.CLW:

1. Press the **Global** button on the Application Tree, to open the **Global Properties** dialog.
2. Press the **Embeds** button.
3. Select the *After Global Includes* embed point and add the following embedded source code:

```
include('prnprop.clw')
```

To change the printer device:

1. From the Application Tree, select your report procedure and press the **Properties** button.
2. Press the **Embeds** button.
3. Select the *ProcessManager Method Executable Code--Open (Priority:2500)* embed point and add the following embedded source code:

```
sav::printer = PRINTER{PROPPRINT:Device}
! save windows default printer
PRINTER{PROPPRINT:Device}='HP Laserjet Series II'!set new default printer
```

The device property takes the name of the printer device. This can be found by looking in windows print manager. The device is the actual printer name. The device property string is case insensitive.

At the end of the report, restore the original default printer:

1. From the Application Tree, select your report procedure and press the **Properties** button.
2. Press the **Embeds** button.
3. Select the *ProcessManager Method Executable Code--Close (Priority:7500)* embed point and add the following embedded source code:

```
PRINTER{PROPPRINT:Device} = sav::printer
```

How to Change Your Application's Dictionary

On occasion, your application development may be split across two or more different data dictionaries. For example, you may want to merge two separate dictionaries into one, then point your applications at the resulting new dictionary.

Do **not** use the **Application ▸ Change Dictionary** menu command to change to a different dictionary. This command is only appropriate when the two dictionaries are ***exactly*** the same.

To continue developing an existing application with a *different* but similar data dictionary, it is necessary to allow the application to re-establish its internal references to dictionary IDENTs (see the *Advanced Programming Guide* PDF for more information on IDENTs). Follow these steps:

1. Make backups of all .APPs and .DCTs involved in the change.

2. Create the modified dictionary (.DCT).

You may cut and paste from other dictionaries, import from text dictionaries (.TXD), or use the Dictionary Editor.

3. Create a text application file (.TXA) referencing the new dictionary.

Load the application to modify into the Clarion environment.

Choose **File ▸ Export Text**.

In the *Save Application Text* dialog, specify the filename and path of the .TXA file.

Using any text editor, edit the .TXA file. The dictionary name usually appears on the third line of the .TXA file, like this: DICTIONARY 'OLD.DCT.' Change this line to show the new dictionary name: DICTIONARY 'NEW.DCT.'

4. Create a new application by importing the .TXA file.

From the Clarion development environment, choose **Setup ▸ Application Options,** then clear the **Require a dictionary** check box.

Close any active applications (press the **OK** button).

Choose **File ▸ New,** then specify the application name and path in the **New** dialog. If you have backed up your application, you may want to specify the same name and path. Press the **Create** button.

From the **Application Properties** dialog, clear the **Application Wizard** check box, then press the **OK** button.

From the Clarion development environment, choose **File ▸ Import**, then specify the .TXA file you created in the previous step. Press **Replace All** when prompted by the *Procedure name clash* dialog.

That's it. You can now develop your application using the new dictionary.

How to Choose Data Types

Dates and Times

LONG is the usual data type for both date and time values where external compatibility is not an issue. This allows you to use Clarion Standard Date and Time arithmetic. It is also a more efficient storage mechanism, since a LONG is 4 bytes of storage while mm/dd/yy in a STRING is 8 bytes.

The DATE and TIME data types are useful only for external compatibility with Btrieve files that already use them. There is no other advantage to DATE and TIME data types than the compatibility issue, since the values they contain are always internally converted to LONG before any math is done on them.

For xBase files, you generally use STRING data types for date storage, because STRING is the actual data storage that all the xBase file systems use. See the specific file driver's documentation in the User's Guide for further information on this issue, because your choice can be affected by whether the file already exists or your program needs to create it.

ZIP Codes

The DECIMAL data type is very good for zip codes because it's a packed decimal format -- a 9 digit ZIP+4 in a DECIMAL is 5 bytes of storage while 9 digit zip in a STRING is 10 bytes. This kind of storage savings is a real consideration when you're setting up a large database. Since there is no math to perform on ZIP codes, storage is generally a larger consideration than performance.

Phone Numbers

The DECIMAL data type is also very good for non-international phone numbers, for the same reasons as ZIP codes. Since you're dealing only with phone numbers in your own country, you should be able to define the exact number of digits and format to display. For U.S. numbers, you can store the area code separately from the phone number -- use a SHORT for the area code (3 digits in 2 bytes) and a LONG for the phone number (7 digits in 4 bytes) -- and achieve the same storage as a single DECIMAL(10,0) (10 digits in 6 bytes).

If your program must deal with international phone numbers, the best data type is a STRING, because the most common method of indicating the country code is with a plus sign (+). For example, +44 (0)800 555 1212 indicates country code 44 (the United Kingdom). You should make the STRING at least 19 characters, since the number of digits in the number can vary from country to country, and even within separate sections of the same country.

"Customer" Numbers

"Customer" Number is defined for this discussion as: any internal number in your program used primarily as the linking field between Parent and Child files.

LONG is the most common data type used for internal numbering for linking purposes. It is very efficient for both storage (4 bytes) and execution (it is one of the base data types used internally by the Clarion libraries – see Base Types in the Language Reference). Any KEY based on a single LONG field is very efficient and small on the disk, since it will require fewer key node splits than a KEY based on a longer STRING (like the "customer" name).

Money

The best data type for any field that will store money values is DECIMAL. Using DECIMAL provides the most efficient storage, since it is a packed-decimal format. It also provides Binary Coded Decimal (BCD) math functionality, which means that calculations are executed in Base 10 instead of Binary (as it would if you use REAL). Using BCD math eliminates the rounding and significant digit problems that you can encounter when you use any of the floating point data types (REAL, SREAL, BFLOAT4, BFLOAT 8).

How to Clip and Concatenate Name Fields

First names and last names are often stored in separate fixed length fields. If printed directly from those fields they usually contain extra spaces and no punctuation, like this: Katie Kelton E. However, you may want the name to look like this: Kelton, Katie E. Follow these steps to display names with punctuation and without the extra spaces. This procedure assumes you already have a report procedure that displays your name fields.

1. Create a Memory Variable to Hold the Concatenated Names.

2. Create an Expression to Clip and Concatenate the Names.

3. Place the Variable in Your Report.

Create a Memory Variable to Display the Concatenated Names

1. From the Application Tree dialog, highlight your report procedure and press the **Properties** button.
2. From the **Procedure Properties** dialog, press the **Data** button.
3. From **Local Data** dialog, press the **Insert** button to add a local variable.
4. From the **New Field Properties** dialog, define the new variable as follows.

In the **Field Name** box, type *FullName*. This is the name by which we will refer to the variable in our concatenation formula, and in our report.

The variable should be long enough to hold all the name fields to be concatenated, plus any punctuation and spaces. For example: if the first name field is 20 characters, the last name field is 20 characters, the middle name field is 1 character and you plan to use a comma, a period, and a space for punctuation, then you will need 20 + 20 + 1 + 3 (ie. 44) characters for your new variable.

In the **Characters** spin box, type 44, then press the **OK** button, then the **Close** button to return to the **Procedure Properties** dialog.

Create an Expression to Clip and Concatenate the Names

You can create the same result by typing the assignment statement into the "Before Print Detail" embed point for your procedure. However, we will use the Formula Editor to accomplish our goal.

1. From the **Procedure Properties** dialog, press the **Formulas** button.
2. In the **Name** field, type *Name Concatenation*.
3. For the **Class** field, press the ellipsis (...) button and choose ***Before Print Detail*** from the Template Class list, then press the **OK** button.

   The class (also called Formula Class) determines *when* the expression is evaluated and the assignment performed.
4. For the **Result** field, press the ellipsis (...) button, highlight ***Local Data***, highlight ***FullName*** (the variable we defined above), then press **Select** the button.
5. Press the **Functions** button, then choose CLIP from the Functions list, then press **OK**.
6. Press the **Data** button, highlight your report procedure file, highlight the last name field, then press **Select** the button.
7. At the insertion point, type the following code:

   ```
   &', '&CLIP(FirstNameField)&' '(MiddleInitialField)&'.'
   ```

   Where *FirstNameField* is the first name field in your report file, and *MiddleInitialField* is the initial field in your report file. Steps 5 - 7 show how you may type your expression, or use the Formula Editor buttons to choose valid operands and operators.
8. Press the **Check** button to check your expression syntax.

   A green check appears if syntax is correct, otherwise a red X appears.
9. Press **OK** to exit the Formula Editor; press **OK** again to exit the Formulas dialog.

Place the Variable in Your Report

1. From the **Procedure Properties** dialog, press the **Report** button.
2. Delete all but one of the name fields from your report.

   CLICK on the field, then press DELETE.

Display the Property Toolbox (Option ▸ Show Propertybox) to help you identify your report fields.

3. CLICK on the remaining name field to select it.

   If you haven't displayed the Property Toolbox as described in the **Tip**, do it now.
4. In the Property Toolbox **Picture** field, type *@S44*.
5. In the Property Toolbox **Use** field, type *FullName*.
6. Resize the field by dragging its handles.

How to Complete an Entry Field when the Last Character is entered

In Clarion's DOS products, an entry field with the immediate attribute (IMM) is automatically completed when the last character was typed. In Clarion, the IMM attribute behaves differently. In Clarion, an entry field with the IMM attribute generates an Event:NewSelection as each character is typed.

To mimic the behavior of Clarion DOS products, a few lines of embedded source code are needed.

In the Window Formatter:

1. RIGHT-CLICK on the control, then select **Properties.**
2. On the **Extra** tab, check the **Immediate** box, then press the **Ok** button.
3. DOUBLE-CLICK on the control to access the **Embedded Source** dialog.
4. Select the *Control Event Handling--Event:NewSelection* embed point for the control, then press the **Insert** button.

This embed point will only appear after you have checked the **Immediate** box.

5. Select SOURCE and type one of the following code segments in the Embedded Source Code Point:

Use this code if the field is a string:

```
UPDATE(?PRE:FieldName)
IF LEN(CLIP(PRE:FieldName)) = SIZE(PRE:FieldName) AND KEYCODE()<> MouseLeft
  ! use size of -1 with CSTRING or PSTRING
  SELECT(?+1)
END !IF
```

Use this code if the field is any non-string numeric data type:

```
UPDATE(?PRE:FieldName)
Str" = PRE:FieldName
IF LEN(CLIP(Str") = 5 AND KEYCODE() <> MouseLeft
  SELECT(?+1)
END                                    !IF
```

In this example the value of the field is assigned to an implicit string variable (Str") so that its length can be determined. Its length is compared to a constant number (in this case 5) instead of using the SIZE function. Since SIZE() returns the number of BYTES in the Use variable, it is not a valid comparison for numeric data types.

5. Set the Priority to 2500.

Note: This example does not handle leading zeros. If your data can contains leading zeros, you will have to modify the embedded source code to handle them.

How to Control Page Breaks

One of the main considerations when laying out a report is unplanned page breaks. For example, you can't always predict how long or short a group will be. You therefore should plan on how your report will behave when it reaches the end of the page, yet there is still more data (in the group) to print.

There are several options available. Each controls what happens when a DETAIL section may be split at the end of a page.

To access the dialogs which allow you to set these options, DOUBLE-CLICK the DETAIL section. Alternatively, select either section and press the **Properties** button. This displays the **Detail Properties** dialog, containing the options below.

You may also set these options for group headers or footers. The options are available in the **Group Header Properties** and **Group Footer Properties** dialogs.

PAGEAFTER

To print the DETAIL, then force a new page, type a value in the **Page after** box in the **Detail Properties** dialog. This sets the PAGEAFTER attribute. This prints the DETAIL, then prints the page FOOTER, then begins a new page.

To print a separate page for each record, place the variable strings and/or controls you wish in the DETAIL, and specify the PAGEAFTER attribute in the Detail Properties dialog.

The page number automatically increments, unless you reset it. To reset the page number to a value you specify, type it in the **Page after** field in the **Detail Properties** dialog.

PAGEBEFORE

To print the DETAIL structure on a new page, type a value in the **Page before** box in the **Detail Properties** dialog. This sets the PAGEBEFORE attribute. The report prints the full DETAIL starting at the top of the next page. The report FOOTER, however, prints on the first page.

The page number automatically increments, unless you reset it. To reset the page number to a value you specify, type it in the **Page before** field in the **Detail Properties** dialog.

WITHNEXT

To prevent 'widow' elements in a printout, type a value in the **Keep next** field in the **Detail Properties** dialog. This sets the WITHNEXT attribute. A 'widowed' print element is one which prints, but then is separated from the succeeding elements by a page break.

The value specifies the number of succeeding elements to print--a value of '1,' for examples, specifies that the next element must print on the same page, else page overflow puts them on the next.

WITHPRIOR

To prevent 'orphan' elements in a printout, type a value in the **Keep prior** field in the **Detail Properties** dialog. This sets the WITHPRIOR attribute. An 'orphaned' print element is one which prints on a following page, separated from its related items.

The value specifies the number of preceding elements to print--a value of "1," for example, specifies that the previous element must print on the same page.

When placing subtotals or totals in a DETAIL, use the WITHPRIOR attribute to insure they print with at least one member of the column above it when a page break occurs.

How to Convert a File--Generate Source

If a file's definition needs to be changed and meaningful data exists, follow these steps to convert the file. This method creates an executable file that you can ship to end users to convert their data files. If you want to convert a file without creating a file conversion program see How to Convert a File (without generating source) .

1. Open (load) the dictionary that contains the file to be modified.

2. Copy the data file definition to a new name. To copy a file definition, highlight the file to be copied in the Files List and press CTRL+C, then press CTRL+V to paste it. You will be prompted to supply a new name and prefix. (Example - copy Customer to OldCustomer)

An alternative would be to copy the entire dictionary to a new name. You might use this method if there are multiple files to be converted in one session. Clarion allows files to be converted from one dictionary to another.

3. After the file definition has been copied, make any necessary changes (add fields, change the file driver type, etc.) to the definition with the original name. In our example above, the Customer file is the file to be modified.

4. Save the Dictionary after file modification and close it.

The Dictionary file **must** be closed in order to use it for file conversion.

5. Load the file in the Database Manager File utility (**File ▸ Open, Database Tab**)

6. Navigate to the file to convert, then select it.

You are prompted to specify a file driver

7. Select the file's driver from the drop-down list.

The Database Manager opens the file and displays it's contents.

8. Choose **File ▸ Convert File** (or press CTRL+V).

The **File Convert** dialog appears, prompting for the information below:

9. In the **Target Filename** field, specify the name of the new file.

This defaults to the current file name. Specify a different name if you have not backed up your data file.

10. In the **Target Dictionary** field, specify the dictionary which contains the file definition to which to convert.

11. In the **Target Structure** field, specify the structure within the target dictionary which defines the target file.

12. In the **Generated Source** field, specify the file name for the generated source code or accept the default of CONVERT.CLW.

Note: The conversion utility writes CONVERT.CLW to the subdirectory of the active application or project file, not necessarily to the subdirectory containing the data or the data dictionary.

13. Press the **OK** button.

This generates a source file and a corresponding project file (.PRJ) which you can now compile and link to create an executable program to perform the file conversion.

14. Press the **Exit** button to close the data file in the Database Manager.

Note: Prior to executing the generated conversion program, you must close the data file open in the Database Manager.

15. Load the conversion program by choosing **File ▸ Open** and selecting the **Source** tab.

16. Select *CONVERT.CLW* (or the file name you specified) in the **File Open** dialog.

17. Edit the source code as required to make the field assignments.

See *Editing Source Code to Make Field Assignments.*

18. Choose **Project ▸ Set** to load the project file.

Navigate to the project file and select it. This defaults to CONVERT.PRJ.

19. Press to Make and Run the conversion program.

After the conversion program runs:

20. Check the file that has just been converted by opening it with the Database Manager.

After viewing the converted file, some clean up steps are all that's left to do:

21. Delete the "old" file definition from the active dictionary, or archive it into a backup dictionary file.

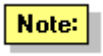
22. If the converted file is located in a different directory, you may now copy it into the working program directory. If you renamed the file, you may rename it to the original file name at this time.

The conversion process is now complete. This example creates CONVERT.EXE which you may be ship to end users to convert their files.

How to Convert a File (without generating source)

If a file's definition needs to be changed and meaningful data exists, follow these steps to convert the file. This method does not create an executable file. It converts the data file on your system to a new format. If you want to create a file conversion program see How to Convert a File--Generate Source.

It is always a good idea to make backup copies of your files before running any conversion process.



If you change the name of a field, you must generate source code, and edit the source code to make the field assignments. Otherwise, your data will be lost.

1. Open (load) the dictionary that contains the file to be modified.
2. Modify the data file definition as desired (add fields or keys, etc.).
3. With the modified file highlighted, choose **File ▸ Browse** to load the data file in the Database Manager.

   A message appears, warning that the physical file structure does not match the dictionary declaration.
4. Press the **Yes** button to convert the file.

The conversion process is now complete!

How to Create a Complex Assignment Expression

An IF structure assigns a value to the **Result** variable based on the true/false evaluation of a *single* logical expression. There are *two* possible assignments. If the condition tested is true, one assignment is made, if not true (false), then the other assignment is made. Nesting IF structures allows for even more alternative assignments.

A CASE structure selectively assigns a value to the **Result** variable based on the evaluation of multiple OF expressions against the CASE expression. The CASE structure offers a less complicated (but less flexible) method for assigning alternative values. CASE structures may also be nested, and IF and CASE structures may be nested within each other. .

Complex Expressions - IF

Use an IF structure to assign one of two values to the **Result** field depending on a condition. Nesting IF structures allows more complex alternative assignments.

To create an IF conditional formula (from the **Procedure Properties** dialog):

1. Press the **Formulas** button.

2. Press the **New** button.

The **Formula Editor** dialog appears.

3. In the **Name** field, type a name for the formula.

4. Press the ellipsis (...) button next to the **Class** field to choose Formula Class.

A formula class determines *where* in the generated source code its calculation is performed. Each Clarion procedure template has its own set of formula classes. For example, in the Form Template there is a class called "After Lookups" which tells the Application Generator to compute the formula after all lookups to secondary files are completed for the procedure.

5. In the **Description** field, type a description of the formula.

6. In the **Result** field, type the variable to which the result of the expression is assigned, or press the ellipsis (...) button to choose a variable from the **Select Field** dialog.

You can choose a local, module, or global variable, or a data dictionary field.

7. Press the **Conditionals** button.

8. Press the **IF..THEN** button.

The IF structure appears in the **Structure** window.

9. On the **Statement** line, enter the IF condition to evaluate.

You can type the expression, or you can use the **Operators** and **Operands** buttons to select expression components, or you can do both.

10. Press the **Check** button to check your syntax.

11. Press the **Accept** button to insert your expression into the structure.

12. Highlight the line below the IF line in the **Structure** window.

This is where the "True" assignment expression goes.

13. On the **Statement** line, enter the "True" assignment expression.

Again, you can type the expression, or you can use the **Operators** and **Operands** buttons to select expression components, or you can do both. If the IF condition is true, this expression is evaluated and the resulting value is assigned to the Result variable.

A "true" assignment expression is *not* required. If no assignment is entered, then no assignment is made.

14. Press the **Check** button to check your syntax.

15. Press the **Accept** button to enter your expression into the structure.

16. Highlight the line below the ELSE line in the **Structure** window

This is where the "False" assignment expression goes.

17. On the **Statement** line, insert the "False" assignment expression.

Again, you can type the expression, or you can use the **Operators** and **Operands** buttons to select expression components, or you can do both. If the IF condition is false, this expression is evaluated and the resulting value is assigned to the Result variable.

A "false" assignment expression is *not* required. If no assignment is entered, then no assignment is made.

18. Press the **Check** button to check your syntax.

19. Press the **Accept** button to insert your expression into the structure.

To add a **nested** control structure:

20. Highlight one of the assignment lines in the **Structure** window.

21. Press either the **CASE..OF** or **IF..THEN** button.

A new nested structure appears in the **Structure** window.

22. Insert expressions on the appropriate lines as described above.

Again, you can type the expression, or you can use the **Operators** and **Operands** buttons to select expression components, or you can do both

23. When your control structure is complete, press the **OK** buttons in the **Conditionals, Formula Editor, and Formulas** dialogs.

Complex Expressions - CASE

A CASE structure can be used to assign one of several values to the **Result** field depending on which OF expression is equal to the CASE expression.

To create a CASE conditional formula (from the **Procedure Properties** dialog):

1. Press the **Formulas** button.

If you already have formulas in the procedure, the **Formulas** dialog appears. If this is the first formula in this procedure, the **Formula Editor** dialog appears, so skip step 2.

2. Press the **New** button.

The **Formula Editor** dialog appears.

3. In the **Name** field, type a name for the formula.

4. Press the ellipsis (...) button next to the **Class** field to choose Formula Class.

A formula class determines *where* in the generated source code its calculation is performed. Each Clarion procedure template has its own set of formula classes. For example, in the Form Template there is a class called "After Lookups" which tells the Application Generator to compute the formula after all lookups to secondary files are completed for the procedure.

5. In the **Description** field, type a description of the formula.

6. In the **Result** field, type the variable to which the result of the expression is assigned, or press the ellipsis (...) button to choose a variable from the **Select Field** dialog.

You can choose a local, module, or global variable, or a data dictionary field. This name appears in the **Formulas** dialog list.

7. Press the **Conditionals** button.

8. Press the **CASE..OF** button.

The CASE structure appears in the **Structure** window.

9. On the **Statement** line, enter the CASE expression that is compared to the multiple OF expressions.

You can type the expression, or you can use the **Operators** and **Operands** buttons to select expression components, or you can do both.

10. Press the **Check** button to check your syntax.

11. Press the **Accept** button to insert your expression into the structure.

12. Highlight the OF line below the CASE line in the **Structure** window.

This is where the first OF *comparison* expression goes.

13. On the **Statement** line, enter the OF *comparison* expression.

Again, you can type the expression, or you can use the **Operators** and **Operands** buttons to select expression components, or you can do both. At runtime, if the CASE expression equals this OF expression, then the subsequent assignment expression is evaluated and the resulting value is assigned to the Result variable.

14. Press the **Check** button to check your syntax.

15. Press the **Accept** button to insert your expression into the structure.

16. Highlight the line below the OF line in the **Structure** window.

This is where the first OF *assignment* expression goes.

17. On the **Statement** line, insert the OF *assignment* expression.

Again, you can type the expression, or you can use the **Operators** and **Operands** buttons to select expression components, or you can do both. At runtime, if the CASE expression equals the above OF expression, then this assignment expression is evaluated and the resulting value is assigned to the **Result** variable.

18. Press the **Check** button to check your syntax.

19. Press the **Accept** button to insert your expression into the structure.

To add additional OF statements:

20. Highlight an OF line in the **Structure** window.

21. Press the **Case..OF** button

22. Insert your expressions in the same manner described above.

To add a nested control structure:

23. Highlight an assignment line in the **Structure** window.

24. Press either the **CASE..OF** or **IF..THEN** button

25. Insert expressions on the appropriate lines following the instructions in the previous sections.

Again, you can type the expression, or you can use the **Operators** and **Operands** buttons to select expression components, or you can do both.

26. When your control structure is complete, press the **OK** buttons in the **Conditionals, Formula Editor, and Formulas** dialogs.

How to Create a Data Dictionary

This section provides an overview of the **general** process of creating a data dictionary. This involves the following steps:

1. Design Your Dictionary and Database.

Planning and organizing your application's database design up front can result in a more efficient application, as well as much shorter development times.

2. Create the Dictionary (.DCT) File.

3. Add Files to the Dictionary .

4. Add Fields to the Files.

5. Define Keys for the Files.

6. Define File Relationships.

Including custom referential integrity constraints for related files.

How to Create a Dictionary (.DCT) File

You generally create a data dictionary as the first step in creating an application. Therefore, you will access it first from the development environment"s main menu.

To open the **Dictionary Editor** to create a new dictionary file:

1. Choose **File** ▸ **New** ▸ **Dictionary** .
2. Specify the path (**Folders**) and **File Name** for your dictionary file, then press the **Create** button.

   The **Dictionary** dialog appears.
3. Press the **Dictionary Properties** button at the bottom of the dialog.
4. On the **Comments** tab, type the description in the space provided.

The description is solely for your convenience, and has no effect on the application. It is useful when other programmers take over your project, or for when you return to the project after a long absence.

Your data dictionary file is created. At this point the dictionary is an empty shell. Use the **Dictionary** dialog to add files, fields, keys, and file relationships.

How to Create a .DLL (Sub-Application)

This section describes the steps to create a program using one main application and several sub-applications compiled and linked as external.DLLs. It is written with Team Development in mind, so it describes some of the aspects of working in a multi-developer environment. Dividing a large project into multiple .DLLs provides many benefits:

- Each sub-application can be modified and tested independently.
- Developers can work on their portion of the project without interfering with others on the development team.
- Each sub-application can be compiled as a DLL and tested in the main program without recompiling the entire project. This reduces compile and link time.
- Dynamic Pool Limits are avoided in large projects.
- Future updates can be deployed by shipping a new .DLL, reducing shipping costs.

With this approach, each Team Member creates a separate .DLL that is called by a "master" application. This requires splitting the application into a "Main" executable and "secondary" .DLLs. The individual team members maintain separate application files for each component. The Team Leader creates a master application that calls the sub-applications and a "data" application that contains (and exports) all the File definitions and Global variables. Optionally, members can call procedures from another member's .DLL.

This method also requires extensive pre-planning of the "division of labor" between the various target files created by the application.

The following outlines a possible implementation of this strategy:

1. Create the data dictionary and set up the workstations as described above.

2. Create a "dummy" application to store and export all data declarations. All Global variables or structures and all file definitions are defined (and exported) in this application. Use the following settings:

In the Application Properties:

Dictionary File: The master dictionary residing on the network.

Target Type: DLL

In the Application's Global Properties:

Generate Global Data as External: OFF

File Control Flags:

Generate All File declarations: ON

External: NONE EXTERNAL

Export All File declarations: ON

3. Team members create their own sub-application .APP files, specifying the dictionary file on the network as the data dictionary, and a directory on the local drive as the default directory for the .APP file. Each team member specifies a different target file using the following settings:

In the Application Properties:

Dictionary File: The master dictionary residing on the network.

Target Type: **EXE** during the design and testing phase
DLL when releasing to the master directory.

Note: Changing the Target Type enables procedures to be exported. Make sure that every procedure that is called by the master application or another .DLL has the Export Procedure check box in the Procedure Properties checked (the check box is only available after changing the target type).

In the Application's Global Properties:

Generate Global Data as External: ON

File Control Flags

Generate All File declarations: OFF

External: ALL EXTERNAL

All Files declared in another .App: ON

Declaring Module: Leave this blank

In the Application's Module Tree:

Choose **Application ▸ Insert Module,** select External DLL, then select the corresponding .LIB for the .DLL containing the data definitions.

One particular .APP creates the executable which launches or calls library functions or procedures in the others. To the end user, this is the .EXE program to start when working with the complete application.

4. Team members synchronize their local directory with an equivalent on the network at the end of each day.

5. Team Members release their compiled and linked .DLLs to the Team Leader.

Each sub-application has a "dummy" frame (not exported) that calls the sub-application's procedures so the Team Member can test the sub-application by compiling it as an .EXE. Once it passes testing, the member compiles it to a .DLL by changing the Application Properties' Target File type to .DLL and releases the file to the Team Leader.

6. The Team Leader copies the released .DLLs into the master directory and creates a master .APP file which calls the entry point procedures in the .DLLs.

The Master .APP is typically just a bare bones application with just a splash screen and a main frame with a menu and toolbar. The .DLLs are called at runtime so you don't need to compile a large Master .EXE. The Master .APP should have the same settings as the sub-applications except that it is always compiled as an .EXE.

The master .APP should have these settings:

In the Application Properties:

Dictionary File: The master dictionary residing on the network.

Target Type: **EXE**

In the Application's Global Properties:

Generate Global Data as External: ON

File Control Flags

Generate All File declarations: OFF

External: ALL EXTERNAL

All Files declared in another .App: ON

Declaring Module: Leave this blank

In the Application's Module Tree:

Choose **Application ▸ Insert Module,** Select External DLL, then select the corresponding .LIB for the .DLL containing the data definitions.

Choose **Application ▸ Insert Module,** Select External DLL, then select the corresponding .LIB for the sub-application .DLL. Repeat this step for each sub-application.

For each procedure the main application calls, edit the ToDo procedure as follows:

Template: External template.

Module name: Select the corresponding .LIB for the DLL drop down list.

If necessary delete any empty generated modules.

7. The Team Leader compiles the master .APP and tests the calls to the DLLs.

8. The Team leader repeats the last step on a periodic basis until all work by all developers is complete, and the entire application can be tested.

How to Create a Function with the Application Generator

To Create a Function with the Application Generator, you must specify both a Prototype and a Return Value in the **Procedure Properties** dialog. The prototype must specify at least a data type for the return value. You must specify a prototype *before* you can specify a return value.

1. In the **Prototype** field in the **Procedure Properties** dialog, type the following prototype:

   ```
   (),LONG
   ```

   This specifies no parameters and a return data type of LONG. The data type may be any valid Clarion language data type.

2. Press the **OK** button to close the **Procedure Properties** dialog.

3. Press the **Properties** button to reopen the **Procedure Properties** dialog.

   This enables the **Return Value** field.

4. In the **Return Value** field, press the ellipsis button (...) to select or define a variable to receive the return value for your function.

You should embed code to assign the appropriate value to the returned variable.

Receiving Return Values from Procedures

Although you may call a function with *Call a Procedure* from the **Actions** tab, this method does not allow you to receive return values. Therefore, you should generally use embedded source or hand code to receive a return value from a function. Following is one way to call a function from a control; however, you may call a function in many ways.

1. Go to the **Actions** tab for the control.

2. In the **When Pressed** drop-down list, choose *No Special Action*.

3. Press the **Embeds** button.

4. In the **Embedded Source** dialog, choose the *Accepted* embed point for *Control Event Handling, after generated code*, then press **Insert**.

5. In the **Select Embed Type** dialog, select SOURCE.

6. In the Text Editor, type your function call, for example:

   ```
   ValueFromFunction = MyFunction()
   ```

   where ValueFromFunction is a local, module, or global variable that receives the value returned by the function.

7. Choose **Exit!** from the menu, then save your embedded source when prompted.

How to Create a Key

Add and edit keys and indexes using the **Field/Key Definition** dialog.

Keys are automatically updated whenever records are added, changed, or deleted. **Index files do not update automatically**. A BUILD statement is required to update an index.

1. Select a file from the list on the **Files** side of the **Dictionary** dialog and press the **Field/Keys** button.

2. In the **Field/Keys Definition** dialog, select the **Keys** tab.

3. Highlight a key (if one exists), then press the **Insert** button.

The **New Key Properties** dialog appears.

4. Type a valid Clarion label in the **Key Name** field.

5. Optionally type a **Description**. This displays in various dialog boxes, including the **File Definition** dialog.

6. Select the **Attributes** tab and check all boxes that are appropriate for the key.

A runtime index allows you to declare an index without specifying the key component fields in the Data Dictionary. The application must define the key component fields at runtime, as the second parameter of the BUILD statement. The application may rebuild the same index file at a later time, specifying different key component fields for the index.

7. Optionally type a valid DOS file name in the **External Name** field, if the file system needs one. Clarion automatically adds the proper file extension.

8. Select the **Fields** tab, then press the **Insert** button.

The **Insert Key Component** list appears.

9. DOUBLE-CLICK a field in the list;

This transfers its name to the **Fields** tab, which indicates the field will be part of the new key. Repeat steps 8 and 9 to add more fields to the key.

10. Press **OK** to close the **New Key Properties** dialog.

The **New Key Properties** dialog appears again, ready to accept additional keys.

11. Repeat steps 4 through 10 to create additional keys for this file.

12. When you are finished adding keys, press **Cancel** to close the **New Key Properties** dialog and return to the **Field/Keys Definition** dialog.

At the end of the process, your keys appear on the **Keys** tab, with their field components arranged in order, one above the other in a tree diagram.

To modify a key, select the key and press the **Properties** button in the **Field/Key Definition** dialog. The **Edit Key Properties** dialog appears. If you selected a key component, the **Fields** tab is on top. If you selected the key, the **General** tab is on top.

How to Create a Multi-Page Form

In the Window Formatter:

1. Place a SHEET control on the form window.

One TAB or page is automatically included in the SHEET. The SHEET structure declares a *group* of TAB controls that offer the user *multiple pages* of controls for a single window. The multiple TAB controls in the SHEET structure define the pages displayed to the user.

2. Place additional TAB controls on the SHEET as required.

The TAB structure declares a group of controls. This group is one of many pages of controls that may be contained within a SHEET structure. The SHEET structure's USE attribute receives the text of the TAB control selected by the user.

The Windows 95 standard to change from tab to tab is CTRL+TAB. Clarion TAB controls follow this standard, both in the development environment and in a compiled application.

3. Place controls on the tabs as required.

Required Fields on Tabbed Dialogs

The REQ attribute behaves differently for tabbed dialogs than for single page dialogs. Because the user has the option of never even selecting secondary tabs (pages), special steps are required to enforce entry of required fields that reside on secondary tabs.

Enforcing required entry fields on Tabbed Dialogs:

- Put all required fields on the first tab; add the REQ attribute to the tab and to the required entry fields.
- Make a "Wizard".
- Embed code at the beginning or end of the procedure that selects all tabs with required fields; add the REQ attribute to the required entry fields and to their parent tabs.

How to Create a New Application File

The first step in creating a new application is to create an .APP file. The .APP file holds the procedures, data, and other properties you define for your application. Optionally create a new directory for the application; whenever you open the .APP file, Clarion will use the directory in which the .APP file resides as the working directory.

1. Create a subdirectory for your application, then return to Clarion.

2. Choose **File ▸ New ▸ Application.**

3. Type a name for the .APP file in the **File Name** field. If you want to use the Quick Start wizard, check the box below the file list.

Type a legal DOS filename. Clarion automatically adds the .APP extension.

4. Press the **Save** button.

The **Application Properties** dialog appears. This dialog allows you to define the essential files for the application.

6. Name the .DCT file the application will use in the **Dictionary File** field, or press the ellipsis (**...**) button to select the file in the **Select Dictionary** dialog.

See How to Create a Data Dictionary for information on creating your application's data dictionary. The **Select Dictionary** dialog is a standard **Open File** dialog.

The Application Generator does **not** require a data dictionary to generate an application, if you **uncheck** the **Require a dictionary** box in the **Application Options** dialog.

7. Optionally rename the first procedure from MAIN to another name of your choice.

You can do so by typing another procedure name from the **First Procedure** field.

8. Choose the **Destination Type** from the drop down list.

This defines the type of target file for your application. Choose from **Executable** (.EXE), **Library** (.LIB), or **Dynamic Link Library** (.DLL).

9. Type a name for the application's .HLP file in the **Help File** field, or use the ellipsis (**...**) button to select the file in the Open File dialog.

The Application Generator allows you to name the help topics in your application without determining that the help file exists. You are responsible for creating a .HLP file that contains the context strings and keywords that you optionally enter as HLP attributes for the various controls and dialogs.

10. Choose the **Application Template** field, type. You may accept the default ABC template, or press the ellipsis (**...**) button to select another (third party template set) in the **Select Application Type** dialog.

The selected application template controls code generation.

11. Optionally, check the **Application Wizard** box to use the wizard to create a complete application based on the selected dictionary and a few answers you specify.

12. Press the **OK** button.

Clarion creates the .APP file, then displays the **Application Tree** dialog for your new application.

How to Create a New Menu

Here are the steps for creating a menu starting from an empty window within the **Window Formatter**.

1. Choose the **Menu** ▸ **New Menu** command.

The **Menu Editor** dialog appears. Only the MENUBAR statement is present.

2. In the **New** group box, press the **Menu** button.

This adds the first MENU statement, its name, and its corresponding END statement, ready for editing.

3. In the **Menu Text** field, type the text you want displayed for this MENU.

The ampersand within the menu text signifies that the character *following* the ampersand is the *accelerator key.* For example, type &FILE, so the end user sees **F**ile.

4. In the **Use Variable** field, type a Field Equate Label.

A Field Equate Label has a leading question mark (?), and you should make it descriptive. You refer to the MENU within executable code by its Field Equate Label.

5. In the **New** group box, press the **Item** button.

This inserts an ITEM between the MENU statement and its END statement. Note that ITEMs are used to execute commands or procedures, whereas MENUs are used to display a selection of other MENUs or ITEMs.

When using the Application Generator, each ITEM you place on a MENU or MENUBAR automatically adds an embed point to the control event handling tree in the Embedded Source dialog. This allows you to easily attach functionality to your ITEMs.

6. In the **Menu Text** field, type the text you want to display for this menu ITEM.

For example, type &OPEN, so the end user sees **O**pen. The ampersand within the ITEM name signifies the character *following* the ampersand is the *accelerator key*.

7. In the **Use Variable** field, type a Field Equate Label.

A Field Equate Label has a leading question mark (?), and you should make it descriptive. For example ?FileOpen shows at a glance the intended purpose of this ITEM: to open a file.

8. In the **Message** field, type theMSG attribute contents.

This message text displays in the status bar (if enabled) when the user highlights this MENU or ITEM.

9. In the **Help ID** field, type either a help keyword or a context string present in a .HLP file.

If you fill in the **Help ID** for a MENU or an ITEM, when the user highlights the MENU or ITEM and presses F1, the help file opens to the referenced topic. If more than one topic matches a keyword, the search dialog appears.

A Help keyword is a word or phrase indexed so that the user may search for it in the *Help* **Search** dialog. When referencing a context string in the **Help ID** field, you must identify it with a leading tilde (~).

10. From the **Actions Tab**, choose *Call a Procedure* from the *When Pressed* drop down list.

The procedure you specify executes when the user selects this ITEM. You may specify parameters to pass and standard file actions (insert, change, delete, or select) if applicable, or you may initiate a new thread. The *procedure* appears as a "ToDo" item in your Application Tree (unless you named a procedure that already exists).

This is one way to add functionality to your ITEM. You may also add functionality by ***Run a Program*** from the drop down list, by embedding source code, or by typing an STD ID in the **STD ID** field.

After following these steps, you have a single MENU called **File**, with a single ITEM called **Open**. To add other ITEMS to the MENU, repeat steps **5** through **10**. To add a second MENU, select the END statement and press the **Menu** button. To add a subMENU, select a MENU or ITEM statement and press the **Menu** button.

11. To finish the menu and return to the **Window Formatter**, press the **Close** button.

How to Create a Report based on a Browse Query

This requires 2 control templates on the browse procedure and an extension template on the report procedure. It also requires a means of passing a value (of the query string) to the report procedure either via a global variable or a passed parameter. The following steps assume that an application containing a Browse procedure and a Report procedure (both utilizing the same file and key) is open.

Declare the Global variable

1. Press the **Global** button, then Press the **Data** button on the Global Properties dialog.
2. Press the **Insert** button and add the following variable:

```
GLO:ReportQueryResult STRING(1024)
!Note this can be as much as 5k chars)
```

3. Press the **OK** button to save the new global variable, then press the **Cancel** button when the Insert New Field Properties dialog appears.
4. Press the **Close** button, and then press the **OK** button on the Global Properties dialog.

On the Browse Procedure:

Add the query button control template.

1. Open the Window Formatter on the Browse Procedure.
2. Choose Populate from the Menu, then select Control Template.
3. Highlight BrowseQBEButton, and press the Select button. Position the cross-hair cursor at the position where you want the Upper left-hand corner of the Query button, and click.
4. Right-click on the Query button, and select Actions from the Popup menu.
5. Press the QBE Options Button, and then press the QBE Class tab. Make a note of the Object Name.

This enables a query on the browse and saves the filter expression we will pass to the report.

Add the BrowsePrint button control template.

1. Choose Populate from the Menu, then select Control Template.
2. Highlight BrowsePrintButton, and press the Select button.
3. Position the cross-hair cursor at the position where you want the Upper left-hand corner of the Print button, and click.

Specify the report to call.

1. RIGHT-CLICK on the new Print button, and select Actions from the Popoup menu.
2. Select the Report procedure to call from the Print Button drop-down list.
3. Press the Embeds button.
4. Highlight Accepted, and press the Insert button. Highlight Source, and press the Select button. Type the following line of code:

```
GLO:ReportQueryResult = QBEn.GetFilter
```

Replace *n* with the appropriate number of the QBE object which you noted earlier.

On the Report Procedure

Note: The report MUST use the same key and filter as the browse!

Add the ExtendProgressWindow control template.

1. On the Procedure Properties dialog Press the Extensions button.
2. Press the Insert button and Highlight ExtendProgrssWindow on the list, and press the Select button.
3. Press the OK button to return to the Procedure Properties dialog.
4. Press the Embeds button, and navigate to the ThisReport.Open in the LocalObjects branch of the tree. Embed the following code in the CODE section at a Priority of 5000:

```
IF GLO:ReportQueryResult
 ThisReport.SetFilter(GLO:ReportQueryResult)
 ThisReport.ApplyFilter()
 GLO:ReportQueryResult = ''  !Clear the var
END
```

Exit back to the Application tree saving your work when prompted.

How to Create a Simple Assignment Expression

A simple assignment evaluates an expression on the right side of the equal (=) sign and assigns it to the variable on the left side of the equal sign. The **Formula Editor** helps you build assignment expressions by providing access to all your valid variable names, plus immediate syntax checking.

- Within the Application Generator, DOUBLE-CLICK a procedure, to open the **Procedure Properties** dialog:

1. Press the **Formulas** button.

2. Press the **New** button.

The **Formula Editor** dialog appears.

3. In the **Name** field, type a name for the formula.

4. Press the ellipsis (...) button next to the **Class** field to choose a formula Class.

A formula's class determines **when** its calculation is performed. Each template has its own set of classes. For example, in the Form Template there is a class called "After Lookups" which tells the Application Generator to compute the formula after all lookups to secondary files are completed for the procedure.

5. Optionally, type a description of the formula in the **Description** field.

6. Press the ellipsis (...) button next to the **Result** field to choose the variable to which the result of the expression is assigned.

You can choose a local, module, or global variable, or a data dictionary field.

7. Create your formula on the **Statement** line.

You may type in the expression, you may use the Formula Editor's buttons, or you may use a combination of the two.

The first component of an expression must be an operand, left parenthesis, or a unary minus (the negative sign).

8. Optionally, press an **Operands** button for the first component of your expression.

9. Optionally, press an **Operator** button for the next component of your expression.

10. Continue adding components to your expression until it is complete.

11. Press the **Check** button to check your syntax.

If the syntax is correct, a large green check mark appears to the left of the statement. If there is any incorrect syntax, a large red X appears.

12. Press the **OK** button.

How to Create a Wizard

A wizard is a window with a "tabless" SHEET control containing one or more TABS. You'll need to write the code to handle the "turning of the pages".

This topic explains one method of creating a wizard using **<<Back** and **Next>>** buttons.

1. Create a procedure using the Window Template.

2. Create two Local Variables (by pressing the Data Button on the Procedure properties dialog).

Label	Data Type	Initial Value
TabNumber	Byte	1
MaxTabs	Byte	number of desired TABs

3. Place a SHEET Control on the Window.

4. Place the desired number of Tabs on the SHEET.

5. Design each TAB.

6. On the **Extra** tab, check the **Wizard** box on the **Sheet Properties** dialog.

This adds the WIZARD attribute to the SHEET control, which hides the "tab" portion of the TAB controls. Waiting to add this attribute until after designing the TABs makes TAB design easier.

7. Place two button controls under the SHEET control.

Use	Text
?Back	<<Back
?Next	Next>>

8. Place a third button control under the SHEET control using either a standard button, a Save Button Control template, or a Close Button control template.

Use	Text
?Finish	Finish

The type of control will depend on the task you intend the wizard to perform. If you are using a Save Button control template, you will need to either call the wizard from a browse or set GlobalRequest=InsertRecord in the *WindowManager Method Executable Code -- INIT* [Priority First] embed point.

9. In the point after *WindowManager Method Executable Code -- INIT--Open The Window (Priority 8250)*

```
HIDE(?Finish)
Select(?sheet1,TabNumber)
Disable(?Back)
```

This hides the **Finish** button and disables the **<<Back** button when the window opens.

10. In the *Control Event Handling--?Back, Accepted* embed point, add this code and set the Priority to 2500:

```
TabNumber -=1
CASE TabNumber
 OF 1
  HIDE(?Finish)
  DISABLE(?Back)
  SELECT(?SHEET1,TabNumber)
 OF MaxTabs
  UNHIDE(?Finish)
  DISABLE(?Next)
  SELECT(?SHEET1,TabNumber)
 ELSE
  HIDE(?Finish)
  ENABLE(?Back)
  ENABLE(?Next)
  SELECT(?SHEET1,TabNumber)
 END
```

This code decrements TabNumber, disables inappropriate buttons, and keeps the Finish button hidden until the final TAB.

11. In the *Control Event Handling--?Next, Accepted* embed point, add this code and set the Priority to 2500:

```
TabNumber +=1
CASE TabNumber
 OF 1
  HIDE(?Finish)
  DISABLE(?Back)
  SELECT(?SHEET1,TabNumber)
 OF MaxTabs
  UNHIDE(?Finish)
  DISABLE(?Next)
  SELECT(?SHEET1,TabNumber)
 ELSE
  HIDE(?Finish)
  ENABLE(?Back)
  ENABLE(?Next)
  SELECT(?SHEET1,TabNumber)
 END
```

This code increments TabNumber, disables inappropriate buttons, and keeps the Finish button hidden until the final TAB.

12. Write the code for the Finish button to accomplish the desired tasks and close the window.

How to Create ABC Compliant Classes

The classes you use with the ABC Templates must be ABC Compliant Classes. That is, the classes must conform to the ABC Library specification as documented in the *ABC Library Reference*

The ABC Templates generate code that refers to the properties, methods, and method parameters documented in the *ABC Library Reference*. If those properties, methods, and parameters are not defined within the classes you specify, the template generated code will not compile. Further, if the classes do not perform as documented, the template generated code probably won't work. The easiest way to create ABC Compliant Classes is to derive classes from the ABC Library--this is what the ABC Templates do. See CLASS in the *Language Reference* for more information on deriving classes.

Requirements for ABC Compliant Classes

- Classes must conform to the ABC Library specification as documented in the *ABC Library Reference*
- The header file containing the CLASS declarations must have the .INC file extension
- The header file (.INC) containing the CLASS declarations must be in Clarion's \LIBSRC directory
- The header file (.INC) containing the CLASS declaration must contain the following comment before compilable code begins:

```
!ABCIncludeFile
```

Note:

There is an optional parameter that can be added to the !ABCIncludeFile comment:

```
!ABCIncludeFile(familyclass)
```

familyclass is a string (no quotes) that contains a Class Name used to populate the %pClassCategory built-in template symbol. If absent, 'ABC' is used (no quotes).

The %pClassCategory is used throughout the templates for debugging, code declarations, and documentation of template prompts.

- The CLASS declarations must have the LINK attribute naming the corresponding implementation (.CLW) files.

Meeting these requirements ensures that your ABC Compliant Classes appear in the Application Builder Class Viewer, the Embeditor, the **Embedded Source** dialog, and that the development environment has full information about your classes. With this information, the development environment can correctly manage embed points and code generation for the compliant classes.

EXTENDS and FINAL

During the processing of the ABC header files, there are two special parameters that can be added to methods defined within Classes that control their presence in the Template Source Embed tree interface (only found in the Application Generator).

These attributes must be preceded with **!,** (exclamation point followed by a comma) and no spaces between. For example:

```
Init            PROCEDURE(BYTE Controls)       !,EXTENDS
Throw           PROCEDURE(SHORT Id),BYTE,PROC  !,EXTENDS
Release         PROCEDURE                      !,FINAL
```

By default, all public methods defined will not be visible in the embed tree. For example:

```
MyMethod        PROCEDURE()
```

The reason for this is to optimize embed processing and eliminate embed points not needed.
If you need to make a public method visible in the embed tree, the EXTENDS symbol is added as follows:

```
MyMethod        PROCEDURE()  !,EXTENDS
```

Virtual methods are always visible in the embed tree. If you want to remove (hide) them from the embed tree , the FINAL symbol is added as follows:

```
MyMethod        PROCEDURE()  !,FINAL
```

In addition to this**, Construct** and **Destruct** are special methods that are automatically called for you when an object is instantiated and destroyed respectively. They will always appear on the embed tree in a different color.

```
Construct       PROCEDURE()
Destruct        PROCEDURE()
```

Finally, methods designated as PRIVATE are *never* visible in the embed tree as they cannot be overridden.

How to Create an MDI Menu

Multiple Document Interface applications make special demands upon a program. Often, the program may support a variety of document windows, each of which has a slightly different set of commands from which the user may select.

Normally in this situation, the programmer writes code to monitor which window is active, then changes the menu and toolbar to reflect the options available to the user. Clarion does this automatically.

To create menus for MDI applications:

1. Create a master menu for the APPLICATION frame window.

Most likely, this will include a File menu and a Help menu, since they contain functions that are available even when no document windows are open.

Clarion's Application Frame procedure template comes with a predefined menu with many of the most common functions already provided for you.

You will use the **Window Formatter's Menu Editor** to create your menus. Be sure to choose the FIRST attribute for the File MENU, and the LAST attribute for the Help MENU from the **Position** drop down list. This ensures that when Clarion merges this global menu with local menus, File and Help will keep their correct positions.

2. *Plan* the additional menus for the child windows.

Can they all share the same menu titles? Do they share many of the same commands? Ideally, *most* of the MENUs and ITEMs can be active in *all* the child windows. If there are only a few commands specific to certain windows, plan on disabling those MENUs and ITEMs in the windows that don't support them, and enabling them in those that do.

3. Create the menu for the first child window.

Again, you will use the **Window Formatter's Menu Editor** to create the menu. Add any window-specific MENUs to the first child window. That is, the window-specific MENUs the application frame lacks--such as Edit, Insert, etc.

Optionally, add a File MENU to the first child window. This is necessary only if the child window needs an ITEM on the File MENU that is not already included on the application's File MENU. For example, adding a Close command might be appropriate. If so, add the File MENU to the first child window. Add the Close ITEM to the File MENU.

Add the Window MENU to the first child window. Window MENUs are standard for most windows programs. A typical Window MENU includes the following ITEMs: "Arrange Icons," "Tile," "Cascade," plus a document (windows) list that displays all open child windows and allows the user to switch between them. In many cases this entire MENU, including the document list, can be implemented with standard ID's (StdID's).

4. Exit the **Menu Editor** and save the menu.

5. Test the interaction of these first two menus.

Do they merge the way you planned? Are the correct selections available for the window with focus? Make any adjustments with the **Menu Editor**.

6. Repeat steps **3** through **5** for other child windows.

How to Create Totals and Calculated Fields on Reports

A total field is a variable STRING control with the SUM attribute added. The AVE, CNT, MAX and MIN attributes similarly create averages, counts, maximum, and minimum fields. These attributes may be added by choosing from the **Total type** drop down list in the **String Properties** dialog.

In general, you place a total field in a page or group FOOTER, so that it can total the records since the beginning of the report, since the beginning of the page, or since the beginning of the BREAK group. However, you can also place a total field in a DETAIL structure to provide a running subtotal. A tally (CNT) field in the DETAIL can number the records as they appear on the report.

To specify a total field, place a variable string control as described in *Specifying Fields to Print*. Then DOUBLE-CLICK on the string control to open the **String Properties** dialog. Finally, choose a total type from the **Total type** drop down list. Choose from **Sum, Average, Minimum, Maximum, Count,** and **Page No.**

Optionally, use the **Reset on** drop down list to reset the total to zero before each page or before each group break.

The **Tallies** list on the **Extra** tab allows you to specify exactly when the total should increment – when specific Detail structures print, or when any group break occurs, or any combination of those.

Sub-totals and Page Totals

Sub-totals are created simply by placing a total field within a page or group footer, and reseting the total to zero at the beginning of each page or group. Use the **Reset** drop down list in the **String Properties** dialog to reset the total to zero before each new page or group.

Grand Totals

In effect, grand totals are simply totals that never reset.

Row Totals

Displaying a row total (or any other calculation) requires two steps: assigning a value to a variable, then displaying the variable value in a string control (see *Specifying Fields to Print* above). There is a good example of this process in Chapter 16 of the *Getting Started* book. See *Making it Work*.

The assignment may be done with the Formula Editor, or by embedding a hand coded assignment statement.

Whether you use the Formula Editor or embedded source, the key point to know is that the assignment should be done just *prior* to the PRINT statement. For formulas, you should choose the **Before Print Detail** class. For embedded source, you should choose the **Before Printing Detail Section** embed point.

How to Customize Procedure Templates

Each procedure template can have multiple default procedures. For example, the Window template can have more than one default procedure--each with its own window design, pre-populated controls and/or control templates, and pre-defined local variables.

Default procedures are customized through the **Template Registry**.

You can also create and modify templates.

1. Choose **Setup ▸ Template Registry.**

The Template Registry lists all registered templates.

2. Highlight the procedure template to customize, then press the **Properties** button.

The **Template Procedure Properties** dialog appears.

3. Press the **Defaults** button.

The Edit Default Procedures dialog appears with a list of the default procedures. All of the procedure templates in the shipped with this product have one default procedure each.

4. Press the **Add** button.

5. Provide a name for the new default procedure in the **New Procedure** dialog, then press **OK.**

A standard **Procedure Properties** dialog appears.

6. Edit the procedure as desired, designing a window, adding controls and/or control templates, adding local variables, etc.

7. Exit by pressing the appropriate **OK** or **Close** buttons.

The next time you add a procedure to an application and select the procedure type, you will be prompted to select the default procedure from which to start.

How to Customize Your Window

Use the **Window Formatter** to visually design window elements--windows, menus, toolbars, list boxes, prompts, entry fields, and other controls--on screen. The **Window Formatter** automatically generates the Clarion language source code that defines these elements.

The **Window Formatter** has five major components that help design your window: the Sample Window, the Controls Toolbox, the Fields Toolbox, the Property Toolbox, and the Align Toolbox.

Using the Window Formatter - A Typical Procedure

Here is the typical process for customizing a new window with the **Window Formatter**:

1. Set the size of the window by dragging the handles so that the sample window is the size you wish.
2. Set other window attributes by using the **Window Properties** dialog.

   RIGHT-CLICK the window and choose **Properties** from the popup menu, or select the window and choose **Edit ▸ Properties.**

   Other attributes include the window caption, whether the window is resizeable, whether the window is scrollable, icons, messages, help files, and cursor types associated with the window, and many others.
3. Close the **Window Properties** dialog.
4. Place controls in the window.
5. Set the properties for each control.
6. Preview the window by choosing **Preview!** from the action bar; repeat steps 1 - 6 to make any necessary adjustments while still in the **Window Formatter**.
7. Choose **Exit!** from the action bar to return to the Application Generator or Text Editor.

How to Define File Relationships and Referential Integrity

Define relationships between files in the **New Relationship Properties** dialog. The relationships for the currently selected file appear in the **Related Files** list on the right side of the **Dictionary** dialog.

1. Select a file from the **Files** list on left side of the **Dictionary** dialog.

2. Press the **Add Relation** button.

The **New Relationship Properties** dialog appears.

3. Select the relationship **Type** from the drop down list: **1:Many** or **Many:1**.

The label for the group box immediately below will change to **Child** or **Parent**, depending on your choice.

4. Select the **Related File** from the drop down list.

5. Select **Primary Key** or **Foreign Key** for the first file from the drop down list at the top right of the dialog.

Clarion automatically changes the label for the drop down list (either **Primary Key** or **Foreign Key**) according to the relationship type.

6. Select the **Primary Key** or **Foreign Key** for the related file, if applicable, from the drop down list immediately below the first drop down list.

7. Press the **Map by Name** button to establish the link between the two keys by matching field names within the two keys, or press the **Map by Order** button to establish the link between the two keys by matching the fields by the order they appear within the two keys. Alternatively, you can map each field manually by double-clicking the field name in the **Field Mapping** list.

The **Field Mapping** lists show the actual links established between the two files.

8. Optionally set Referential Integrity Constraints by choosing from the **On Update** and **On Delete** drop down lists in the **Referential Integrity Constraints** group box.

See the section below for further information on Referential Integrity Constraints.

9. Press the **OK** button.

Setting Referential Integrity Constraints

By setting referential integrity constraints in the data dictionary, you can instruct the Application Generator on how to set up executable code for linked field updates and deletions when working with related files.

Referential Integrity requires that a foreign key must always have a match in the primary key. This raises potential problems when the end user wishes to change or delete the primary key record.

The **New Relationship Properties** dialog allows you to specify how the executable code should handle these situations when one of several related records is updated or deleted..

No Action	Instructs the Application Generator **not** to generate any code to maintain referential integrity.
Restrict	Tells the Application Generator to prevent the user from deleting or changing an entry, if the value is used in a foreign key. For example, if the user attempts to change a primary key value, the generated code checks for a related record with the same key value. If it finds a match, it will not allow the change.
Cascade	Tells the Application Generator to update or delete the foreign key record. For example, if the user changes a primary key value, the generated code changes any matching values in the foreign key. If the user deletes a parent record, the code deletes the children too.
Clear	Instructs the Application Generator to change the value in the foreign key to null or zero.

How to Define Procedure Formulas

Creating conditional expressions with the **Formula Editor** actually creates structures in the source code. There are three structures you can create with the **Formula Editor**--a simple assignment expression, or an IF or a CASE structure. You can also nest either of these structures, creating complex conditional statements.

The **Formula Editor** creates simple (unconditional) assignments, the **Conditionals** dialog creates complex conditional assignments, and the **Formulas** dialog manages these formulaic assignments for your procedure. See:

Usually, you'll want to display the result of your assignment in a string control in a window or report. To display it in a window:

1. Press the **Window** button to open the **Window Formatter**.
2. Select the string tool from the **Controls** toolbox (or choose **Control ▸ String**), and CLICK in the window to place the control.
3. With the new string control selected, choose **Edit ▸ Properties** (or RIGHT-CLICK on the string and choose **Properties** from the popup menu).
4. Check the **Variable String** check box in the **String Properties** dialog.

   The **Select Field** dialog appears.
5. Highlight the variable which contains the **result** of your formula and press the **Select** button.

   The **String Properties** dialog returns, note however, that the **Parameter** field has now been replaced by the **Picture** field.
6. Type a valid display picture in the **Picture** field.
7. Press the **OK** button to close the **String Properties** dialog.
8. Choose **Exit!** to close the **Window Formatter** and return to the **Procedure Properties** dialog.

How to Design Your Dictionary and Database

This topic provides a quick review of relational database theory. Planning and organizing your application's database design up front can result in a more efficient application for the end user, not to mention saving hours of redesign later.

The relational model concerns itself with three aspects of data management: **structure**, **integrity,** and **manipulation**. For our purposes, we will discuss the three practical requirements of these aspects: data normalization, keys, and relational operations.

Normalization

At its simplest, data normalization means that a data item should be stored at only one location. There are two benefits to this: lowered disk space requirements, and easier data maintenance. To achieve this end, a relational database design splits the data into separate, related files. For example, assume a very simple order-entry system which needs to store the following data:

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

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

```
Customer File: Customer Name
               Customer Address

Order File:    ShipTo Address
               Order Date

Item File:     Product Ordered
               Quantity Ordered
               Unit Price
```

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

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

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

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

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

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

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

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

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

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

File Relationship

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

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

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

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

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

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

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

Many-to-Many is the most difficult file relationship with which to deal. It means that multiple records in one file are related to multiple records in another file. Expand the previous example to fit a manufacturing concern which buys Parts and makes Products. One Part may be used in many different Products, and one Product could use many Parts.

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

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

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

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

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

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

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

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

Translating the Theory to Clarion

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

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

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

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

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

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

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

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

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

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

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

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

Referential Integrity

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

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

1) What do you do when the user wants to delete the Primary Key record?

2) What do you do when the user wants to change the Primary Key value?

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

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

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

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

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

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

Summary

Each data item should be stored once.

Separate files are used to eliminate data duplication.

Files are related by Primary and Foreign Keys.

A Primary Key is a unique (and non-null) index into a file which provides for individual record access.

A Foreign Key contains a reference to the Primary Key of some other file.

One-to-Many file relationships are the most common. They are also referred to as Parent-Child and Many-to-One (same relationship, reverse view).

One-to-One file relationships are most commonly created to hold data that is not always needed in every record.

Many-to-Many relationships require a "Join" file which acts as a broker between the two files. The Join file inserts two One-to-Many relationships between the Many-to-Many relationship.

Only those Primary and Foreign Keys that the application needs (as a practical consideration) for specific access to the files need to have Clarion KEYs declared.

Referential Integrity means that all Foreign Keys contain valid references to Primary Keys.

Maintaining Referential Integrity requires executable code that tests for Update or Delete of the Primary Key values.

The three common solutions to maintaining Referential Integrity are: Restricting (update/delete not allowed), Cascading (also update/delete the Foreign Key), or Nullifying the Foreign Key (assign null values to the Foreign Key).

How to Display the Sort Field First on a Multi-Key Browse

Clarion's Browse Wizard generates multi-key browses for files with multiple keys. To see records in a different sort order, the user simply selects a tab with a different key. However, when switching to a new sort order, the sort column does not automatically appear as the first (or leftmost) column in the list box. This example uses the embed points from the Clarion (Compatibility) Templates.

To dynamically shift the sort column to the leftmost position in the list box, follow these steps:

1. Find the FORMAT string for the affected list box and copy it to the clipboard.

In the Application Tree dialog, DOUBLE-CLICK on your browse procedure.

In the Procedure Properties dialog, press the ellipsis (**...**) button next to the **Window** button.

On the LIST control declaration statement, highlight the entire FORMAT attribute parameter and choose **Edit ▸ Copy** to copy it to the clipboard.

Exit! without saving.

2. Paste the FORMAT string into the "Control Event Handling, before generated code; ?CurrentTab; NewSelection" embed point of the Browse procedure.

Press the **Embeds** button.

DOUBLE-CLICK on the "Control Event Handling, before generated code; ?CurrentTab; NewSelection" embed point.

Choose SOURCE from the **Select embed type** dialog.

When the Text Editor opens, choose **Edit ▸ Paste**.

3. "Chop up" the FORMAT string so there is only one list box column definition per line.

Each column definition begins with the width of the column immediately followed by a justification letter (L, R, C, or D). If necessary, you can get the widths and justification for each column from the List Box Formatter.

On each line you need to add enclosing single quotes.

On each line except the last, you need to add a trailing ampersand (&) and pipe (|). The ampersand is the concatenation operator, and the pipe is the line continuation character.

Your code should look something like this:

```
'16L|M~Cust Number~@N4@'     & |
'80L|M~Last Name~@S20@'      & |
'80L|M~First Name~@S20@'     & |
'12L|M~Area Code~@S3@'       & |
'32L|M~Phone Number~@S8@'    & |
'32L|M~Description~@S8@'
```

4. Add explicit QUEUE field numbers to each list box column definition.

Queue numbers are integer constants surrounded by pound (#) signs. The QUEUE field numbers start with 1 and continue in ascending sequence. Your code should now look something like this:

```
'16L|M~Cust Number~@N4@#1#'     & |
'80L|M~Last Name~@S20@#2#'      & |
'80L|M~First Name~@S20@#3#'     & |
'12L|M~Area Code~@S3@#4#'       & |
'32L|M~Phone Number~@S8@#5#'    & |
'32L|M~Description~@S8@#6#'
```

5. Add a CASE structure and property assignment to the embedded source code.

Type *?BROWSE:1{PROP:Format}=* before the first column definition. This creates the assignment statement that formats the list box at run time.

Add a *CASE CHOICE(?CurrentTab)* statement before the first column definition.

Add an *OF 1* statement before the first column definition.

Your code should now look something like this:

```
CASE CHOICE(?CurrentTab)
OF 1             !Tab1, sorted by Cust Number
  ?BROWSE:1{PROP:FORMAT} ='16L|M~Cust Number~@N4@#1#'     & |
                          '80L|M~Last Name~@S20@#2#'      & |
                          '80L|M~First Name~@S20@#3#'     & |
                          '12L|M~Area Code~@S3@#4#'       & |
                          '32L|M~Phone Number~@S8@#5#'    & |
                          '32L|M~Description~@S8@#6#'
```

6. Duplicate the FORMAT string, with the sort column first, for each OF in the CASE.

You should have a separate OF clause for each tab (sort key) in the browse. Cut and Paste the FORMAT string for each OF assignment so that the columns appear in the sequence you want them.

Your code should now look something like this:

```
CASE CHOICE(?CurrentTab)
OF 1              !Tab1, sorted by Cust Number
  ?BROWSE:1{PROP:FORMAT} ='16L|M~Cust Number~@N4@#1#'     & |
                          '80L|M~Last Name~@S20@#2#'      & |
                          '80L|M~First Name~@S20@#3#'     & |
                          '12L|M~Area Code~@S3@#4#'       & |
                          '32L|M~Phone Number~@S8@#5#'    & |
                          '32L|M~Description~@S8@#6#'
OF 2              !Tab2, sorted by Last Name
  ?BROWSE:1{PROP:FORMAT} ='80L|M~Last Name~@S20@#2#'      & |
                          '80L|M~First Name~@S20@#3#'     & |
                          '16L|M~Cust Number~@N4@#1#'     & |
                          '12L|M~Area Code~@S3@#4#'       & |
                          '32L|M~Phone Number~@S8@#5#'    & |
                          '32L|M~Description~@S8@#6#'
END
```

In this example, notice that when the user selects tab 2, QUEUE field number #2# becomes the first column in the FORMAT string, and will be the leftmost column in the list box!

7. **Exit!** the Text Editor and save your changes.

How to Distribute Your Applications

Choosing a Configuration

This section is included to help you decide what kind of target file to specify for your project..

Clarion produces executable files which you may distribute on a royalty-free basis. The applications you distribute require a Windows 32-bit environment (95/98, NT, 2000, XP, etc.).

Clarion executables come in two flavors: .EXE files, and .DLL files. An .EXE file is simply an executable program. A .DLL (Dynamic Link Library) file is executable code that is linked into an .EXE file *at run-time*. This is in contrast to .OBJ and .LIB files which are linked into an .EXE at *compile* time. The most obvious benefit of the .DLL is that it provides a method of modifying .EXE operation without remaking (compiling and linking) the .EXE.

Clarion executables may be distributed in the following four configurations where [x] indicates the 32-bit runtime library for applications running on 32-bit operating systems (95/98, NT, 2000, XP, etc.):

- *.EXE (available only in Professional and Enterprise Editions)

  A one-piece .EXE will usually be larger than an .EXE distributed with .DLLs. However, the one-piece .EXE will probably be smaller than the combined sizes of an .EXE and its associated .DLLs.

  The one-piece .EXE is made as small as possible by Clarion's smart linking process that only links in procedures actually called by the application program (whereas the .DLL contains a fixed set of procedures, whether or not they are actually called by your program).

  A one-piece .EXE cannot have conflicts or problems that arise from linking with the wrong .DLLs at run time.

  Make (compile and link) time for a one-piece .EXE is greater than for an .EXE combined with .DLLs.

- *.EXE + C60RUN[x].DLL

  Splitting the executables between .EXEs and .DLLs allows for more efficient use of disk space. Many Clarion applications (.EXEs) can share a single C60RUN[x].DLL. Or, a single application suite with several .EXEs can share a single C60RUN[x].DLL. However, as a developer, you must ensure that your application accesses the correct version of C60RUN[x].DLL.

  An example of .DLL usage is the typical accounting system where the .EXE controls the system main menu, and calls system subparts such as Accounts Receivable and Accounts Payable from separate .DLLs. This method of distribution allows for program parts to be sold and maintained separately.

Tip

Splitting executables between .EXEs and .DLLs allows for more efficient use of disk space, but less efficient use of RAM. This is because Windows 95 loads an additional C60RUN[x].DLL into memory for each active Clarion executable, and because the C60RUN[x].DLL contains some procedures your .EXE never calls.

Tip

To make an .EXE + C60RUN[x].DLL: in the Global Options dialog, set Target Type to .EXE and set Run-Time Library to *Standalone*.

- *.EXE + C60RUN[x].DLL + *.DLL$_1$ + ... + *.DLL$_n$

  This configuration offers the same advantages and disadvantages as the .EXE + C60RUN[x].DLL configuration. It is listed here to illustrate that you are not limited to a single .DLL, nor are you limited to Clarion .DLLs. Your Clarion applications may make use of .DLLs compiled from other languages as well as the C60RUN[x].DLL and SoftVelocity database driver .DLLs. See the *Programmer's Guide-Database Drivers* for more information on database drivers.

- *.EXE + *.DLL$_1$ + ... + *.DLL$_n$

  This configuration offers most of the same advantages and disadvantages as the .EXE + C60RUN[x].DLL configuration. It is listed here to illustrate that the C60RUN[x].DLL may be linked into another .DLL. This technique "hides" the C60RUN[x].DLL and ensures that your application will never get the wrong version of C60RUN[x].DLL, because, technically, it isn't looking for C60RUN[x].DLL.

Installing and Accessing Your Application's DLLs

If you distribute C60RUN[x].DLL, it must reside in the same directory as the application program, in the Windows\System subdirectory, or in a directory referenced in the system PATH. We recommend that you install C60RUN[x].DLL to the application directory.

Remember, multiple Clarion applications may use the same C60RUN[x].DLL file, thus saving space on the users' hard drive. On the other hand, sharing a single C60RUN[x].DLL raises the possibility of conflicts among applications developed under different versions of Clarion. To avoid possible conflicts, install a separate C60RUN[x].DLL to each application directory, or distribute the application as a single .EXE file, or link the C60RUN[x].DLL into another .DLL that is unique to your application.

The Ship List

For generated applications, Clarion templates automatically create a ship file (.SHP) that contains the names of the files that are needed to run your application. The file is called *application.*SHP and is in the same subdirectory as your .APP file.

This ship file only includes those files that are visible to the templates. Any DLLs loaded in EMBEDs or INCLUDE files may not be visible to the templates, and may not be in the list.

In the case of external library modules, the .LIB file is also included in the list. Some of the .LIBs (WINDOWS.LIB for example) do not have associated DLLs; however, most do have associated .DLLs that you will need to distribute with your application.

In the case of an external library module generated by Clarion, you must ensure that all files on the shipping list for that LIB/DLL are also included.

You can modify your application's ship list by embedding text at the *Inside the Shipping List* Global Embed point.

How to Handle Dates before 1900 and beyond 2000 in the Same Procedure

When your program lets users input dates, it should do so as intelligently as possible. If your program is intelligent and flexible in accepting dates, your end users will appreciate it. For example, forcing your end users to input dates in a rigid dd/mm/yyyy format (or any fixed format) not only slows them down, it can lead to cursing and other expressions of frustration when dates are misinterpreted or rejected altogether.

The keys to intelligent date handling with Clarion are the entry field picture token and Clarion's automatic date parsing functions. Date Notation picture tokens let you display dates in a number of different formats. Better still, date picture tokens in entry fields automatically invoke Clarion's run-time date parsing functions, so you can enter '21' and Clarion expands it to the 21st day of the current month and year. Or you can enter 'DEC' and Clarion expands it to the 1st day of December of the current year. Clarion then formats the expanded date according to the picture token.

The MASK attribute (Entry Patterns check box) on a window preempts the automatic date parsing functions and forces the user to enter the date in the form of the date picture.

To illustrate, let’s look at two different examples: a field containing employee birthdates and a field containing commercial real estate lease expiration dates.

Birth dates from 1900 to 2000

The birth date field is likely to contain dates ranging from the early 1900's to about 1980, and, since your application may be in use for another 10 years, you should probably extend the upper end of this range to 1990 or even 2000.

You want to let your end users input the year of birth without having to explicitly specify the century. For example, in 1997, your program should interpret an entry of 1/1/15 as January 1, 1915 and not as January 1, 2015.

By default, Clarion assumes that a year specified without the century falls within the 100 year period from today-80 years to today+19 years. Thus in 1997, Clarion assumes dates entered fall between 1917 and 2016. This default, as illustrated by our example entry (1/1/15) is not appropriate for your birthday field. However, you can use a date notation picture token to change the default to fit your particular application.

Note:

The assumed 100 year period is always relative to today's date. Therefore, a picture token that is valid for employee birthdays today will also be valid 10 years from today.

To specify an intelligent assumed date range to handle employee birthdays, simply add <99 to your preferred date picture token:

```
@D1<99
```

You may specify the picture token by typing it into the **Picture** field in the Entry Properties dialog, or by pressing the **Picture** field's ellipsis (...) button to open the Edit Picture String dialog. In the **Edit Picture Sting** dialog, choose your preferred date format, then set the **Two digit date range** to 99.

Now, for this field, Clarion assumes that a year specified without the century falls within the 100 year period between today-99 years and today+1 year. So, for birthdays entered in 1997, Clarion assumes the birth year falls between 1898 and 1998, and for birthdays entered in 2007 (10 years from now), Clarion assumes the birth year falls between 1908 and 2008.

Lease Expirations from 1997 to 2096

The lease expiration field is likely to contain dates ranging from the current year (1997) to 99 years in the future (2096).

You want to let your end users input the lease expiration year without having to explicitly specify the century. For example, in 1997, your program should interpret an entry of 1/31/55 as January 31, 2055 and not as January 31, 1955.

By default, Clarion assumes that a year specified without the century falls within the 100 year period from today-80 years to today+19 years. Thus in 1997, Clarion assumes dates entered fall between 1917 and 2016. This default, as illustrated by our example entry (1/31/55) is not appropriate for your lease expiration field. However, you can use a date notation picture token to change the default to fit your particular application.

Note:

The assumed 100-year period is always relative to today's date. Therefore, a picture token that is valid for lease expirations today will also be valid 10 years from today.

To specify an intelligent assumed date range to handle lease expirations, simply add <0 (or >99) to your preferred date picture token:

```
@D1<0
```

You may specify the picture token by typing it into the **Picture** field in the Entry Properties dialog, or by pressing the **Picture** field's ellipsis (...) button to open the Edit Picture String dialog. In the **Edit Picture Sting** dialog, choose your preferred date format, then set the **Two digit date range** to 0.

Now, for this field, Clarion assumes that a year specified without the century falls within the 100 year period between today-0 years and today+99 years. So, for lease expirations entered in 1997, Clarion assumes the year falls between 1997 and 2096, and for lease expirations entered in 2007 (10 years from now), Clarion assumes the year falls between 2007 and 2106.

How do I Handle an Error 47?

Error 47--Invalid Record Declaration indicates that the application's file declaration does not match the data file.

To correct the problem, you should convert the file using one of these two conversion methods:

How to convert a file--Generate Source

How to Convert a File (without generating source)

How to Hide a Window

Sometimes you need to hide a window as you can hide a control. This is an undocumented feature since there is no HIDE attribute on a WINDOW, but it is used internally by the IDE so it's likely to stay:

1. Assign a value to PROP:Hide for the window like this:

```
TARGET{PROP:Hide} = 1
```

This hides the current TARGET window.

How to highlight all the Text in a Control when it gets Focus by a Mouse Click

When you tab to an entry control with the tab key, the entire entry field plus any existing text, is highlighted. Then, when you begin typing, the highlighted text is immediately overwritten. However, when you select an entry control with the mouse, you get an I-beam insertion point somewhere inside the entry field, and existing text is not highlighted.

To force a mouse selection to behave like the tab key, add MouseLeft to the list of Alert keys for the control, then embed some code to "select" the entire field when the mouse is CLICKED:

1. From the **Window Formatter**, RIGHT-CLICK the control and choose **Alert** from the popup menu.
2. From the **Alert Keys** dialog, press the **Add** button.
3. From the **Input Key** dialog, CLICK on **Left Button** in the **Mouse** group.
4. Press **OK** twice to return to the **Window Formatter**.
5. From the **Window Formatter**, RIGHT-CLICK the control and choose **Embeds** from the popup menu.
6. Add the following code to the EVENT:AlertKey embed point for the control:

```
IF KEYCODE() = MouseLeft
  SELECT(?,LEN(?{PROP:ScreenText}))
  ?{PROP:SelStart} = 1
  ?{PROP:SelEnd} = LEN(?{PROP:ScreenText})
END
```

How to Implement Print Preview on a Report Procedure

Generate a report procedure with the Report template or the Report Wizard. Then, check the **Print Preview** box in the "other" **Report Properties** dialog.

Although nearly all font types used on reports will print accurately, for the best display of reports in the Print preview window, use a True Type font (Example – Arial 10pt).

If you prefer to hand code your print preview process, see PREVIEW in the *Language Reference*, for more information and examples.

How to Implement Standard Windows Behavior

There are some menus and commands that you see in almost every windows program. For example, Cut, Copy, and Paste. Clarion provides an easy method for implementing these standard actions in your application menus--with the **Std ID** field on the **Menu Editor** dialog.

Simply enter one of the equates listed below in the **Std ID** field. Clarion will automatically implement the command using standard windows behavior; you do not need any other support for it in your code. The standard equate labels and their associated actions are also contained in the ..\LIBSRC\EQUATES.CLW file.

STD:PrintSetup	Printer Options Dialog.
STD:Close	Closes active window.
STD:Undo	Reverses the last editing action.
STD:Cut	Deletes selection, copies to clipboard.
STD:Copy	Copies selection to clipboard.
STD:Paste	Pastes clipboard contents at the insertion point.
STD:Clear	Deletes selection.
STD:TileWindow	Arranges child windows edge to edge.
STD:TileHorizontal	Arranges child windows edge to edge.
STD:TileVertical	Arranges child windows edge to edge.
STD:CascadeWindow	Arranges child windows so all title bars are visible.
STD:ArrangeIcons	Arranges iconized child windows.
STD:WindowList	Adds child window names to menu.
STD:Help	Opens .HLP file to the contents page.
STD:HelpIndex	Opens .HLP file to the index.
STD:HelpOnHelp	Opens Microsoft's .HLP file for the Windows Help system.
STD:HelpSearch	Opens Microsoft's Help Search utility for the .HLP file.

How to Import a File Definition From an Existing Data File

The Dictionary Editor allows you to quickly add a data file to the dictionary by creating a data definition based on an existing data file.

1. With the **Dictionary** dialog active, select **File ▸ Import File**.

The **Select File** drive dialog appears.

2. Pick a file driver from the drop down list, and press the **OK** button.

Pick the driver of the file whose definition you are creating. The **Open File** dialog appears.

3. Press the ellipsis (...) button and pick the file using the standard Open File dialog.

4. Press the **OK** button twice to close the dialogs.

The Dictionary Editor creates the file definition and the **Edit File Properties** dialog appears.

5. Make any changes to your new file definition, then press the **OK** button.

The data file is added to the dictionary, along with its field and key definitions.

How to Link External Resources

The Project System allows you to specify external resources to link into the executable. These include graphics, such as .BMP, .ICO and .WMF files. By linking them into the executable, you can avoid having to ship them as separate, external files.

If you directly reference a graphic file within a data structure, the compiler automatically links the graphic, so there is no need to add the graphic file to your Project Tree. For example, if you place an IMAGE control in a window, and specify a file by name in the **Image Properties** dialog, the linker automatically includes that file in your executable. But if you assign a different graphic to a control using a runtime property assignment statement, the linker will only include the new file in your executable if you add the file to your Project Tree.

To add graphic files to the executable:

1. Highlight **Library and object files** and CLICK on the **Add File** button.

Select the bitmap, icon, or metafile graphic from the standard Open File dialog.

2. Press the **OK** button to return to the **Project Editor** dialog.

3. Highlight the source code file that references the graphic, and CLICK on the **Edit** button.

The source code file is opened by the Text Editor.

4. Place a tilde (~) in front of the graphic file name in the source code assignment statement (not in data section).

For example: change ?Image{PROP:Text} = 'I.ICO' to ?Image{PROP:Text} = '~I.ICO.' The tilde indicates the program should find the item as a linked in resource, not as an external file.

Optionally, choose **Search** ▸ **Find** to locate the file name.

5. Choose **File** ▸ **Exit**, then CLICK on **Yes** when asked if you want to save.

Now, when you recompile and link, the executable will no longer require the external graphic file.

How to Make a Field Assignment

The File Conversion Utility creates source code to convert a file to a different specification. The conversion is handled automatically except in two cases:

- If a field's label is changed
- If a field is split into two separate fields.

The File Conversion Utility creates source code to convert a file to a different specification. The conversion is automatic except in the following cases:

- If a field's label is changed
- If a field is split into two separate fields.
- If two or more fields are combined.
- If a date is converted to a Clarion standard date from some other file system date format.

In these cases you must modify the source code to handle the field assignments. The portion of the source code you need to examine is the *AssignRecord ROUTINE*. This is where field assignments are made. Here is an example:

```
AssignRecord      ROUTINE
                  CLEAR(CUS:Record)
                  CUS:NUMBER = IN::NUMBER
                  CUS:FIRSTNAME = IN::FIRSTNAME
                  CUS:LASTNAME = IN::LASTNAME
                  CUS:ADDRESS = IN::ADDRESS
                  CUS:CITY = IN::CITY
                  CUS:STATE = IN::STATE
                  CUS:ZIP = IN::ZIP
                  CUS:PHONENUMBER = IN::PHONENUMBER
                  CUS:ENTRYDATE = IN::ENTRYDATE
```

If you examine the source code, you'll see that the first line in the routine clears the record buffer. Next each field in the output file is assigned the value from the matching field in the input file.

Changing a Field's Label

However, if the field labels do not match, no assignment is made. For example, if you change the LastName field to Surname, the File Conversion utility generates a comment to alert you of an assignment you may need to make:

```
AssignRecord      ROUTINE
       CLEAR(CUS:Record)
       CUS:NUMBER = IN::NUMBER
       CUS:FIRSTNAME = IN::FIRSTNAME
       ! CUS:SURNAME = ''
       CUS:ADDRESS = IN::ADDRESS
       CUS:CITY = IN::CITY
       CUS:STATE = IN::STATE
       CUS:ZIP = IN::ZIP
       CUS:PHONENUMBER = IN::PHONENUMBER
       CUS:ENTRYDATE = IN::ENTRYDATE
```

To assign the values from the original file, edit the line containing the assignment to assign the value of LastName to the SurName field as shown below:

```
       CUS:SURNAME = IN::LASTNAME
```

Splitting a Field into Two Fields

Writing the assignment statements to split the contents of a field into two fields involves a little more work, but string slicing minimizes the effort. For this example, let's assume you had a single field in the original file for a phone number and area code. You now want to store the area code in one field and the phone number in another. Assuming that these fields are numeric data types, you need to temporarily assign the value to a string, then "slice" the string to assign the desired portion to each new field.

In this example the original PhoneNumber field is a ten-digit number, the area code is a three-digit number, and the new PhoneNumber field is a seven-digit number. The AssignRecord ROUTINE in the generated file conversion source code looks like this:

```
AssignRecord      ROUTINE
       CLEAR(CUS:Record)
       CUS:NUMBER = IN::NUMBER
       CUS:FIRSTNAME = IN::FIRSTNAME
       CUS:LASTNAME = IN::LASTNAME
       CUS:ADDRESS = IN::ADDRESS
       CUS:CITY = IN::CITY
       CUS:STATE = IN::STATE
       CUS:ZIP = IN::ZIP
       ! CUS:AREACODE =
       CUS:PHONENUMBER = IN::PHONENUMBER
       CUS:ENTRYDATE = IN::ENTRYDATE
```

Notice there is an assignment from the original PhoneNumber field to the new PhoneNumber field. However, since the new field only stores seven digits, you should change this. To handle the field assignments, create an implicit string variable, assign to it the value of the original PhoneNumber field, then use string slicing to assign the desired portions to the new fields, as shown below:

```
AssignRecord      ROUTINE
       CLEAR(CUS:Record)
       CUS:NUMBER = IN::NUMBER
       CUS:FIRSTNAME = IN::FIRSTNAME
       CUS:LASTNAME = IN::LASTNAME
       CUS:ADDRESS = IN::ADDRESS
       CUS:CITY = IN::CITY
       CUS:STATE = IN::STATE
       CUS:ZIP = IN::ZIP
       TempPhoneNumber" = IN::PHONENUMBER
       CUS:AREACODE = TempPhoneNumber"[1:3]
       CUS:PHONENUMBER = TempPhoneNumber"[4:10]
       CUS:ENTRYDATE = IN::ENTRYDATE
```

For more information on String Slicing, see Implicit String Arrays and String Slicing.

Combining Two or More Fields

In this example the original PhoneNumber field is a seven-digit number, the AreaCode is a three-digit number, and the new PhoneNumber field is a ten-digit number. The AssignRecord ROUTINE in the generated file conversion source code looks like this:

```
AssignRecord      ROUTINE
       CLEAR(CUS:Record)
       CUS:NUMBER = IN::NUMBER
       CUS:FIRSTNAME = IN::FIRSTNAME
       CUS:LASTNAME = IN::LASTNAME
       CUS:ADDRESS = IN::ADDRESS
       CUS:CITY = IN::CITY
       CUS:STATE = IN::STATE
       CUS:ZIP = IN::ZIP
       CUS:PHONENUMBER = IN::PHONENUMBER
       CUS:ENTRYDATE = IN::ENTRYDATE
```

Notice there is an assignment from the original PhoneNumber field to the new PhoneNumber field, but the original AreaCode is omitted. To handle the field assignments, create two implicit strings and assign them the original AreaCode and PhoneNumber values, then concatenate the strings and assign the result to the new PhoneNumber as shown below:

```
AssignRecord      ROUTINE
       CLEAR(CUS:Record)
       CUS:NUMBER = IN::NUMBER
       CUS:FIRSTNAME = IN::FIRSTNAME
       CUS:LASTNAME = IN::LASTNAME
       CUS:ADDRESS = IN::ADDRESS
       CUS:CITY = IN::CITY
       CUS:STATE = IN::STATE
       CUS:ZIP = IN::ZIP
       TempAreaCode" = IN::AREACODE
       TempPhoneNumber" = IN::PHONENUMBER
       CUS:PHONENUMBER = CLIP(TempAreaCode") &
TempPhoneNumber"
       CUS:ENTRYDATE = IN::ENTRYDATE
```

For more information on String Slicing, see Implicit String Arrays and String Slicing .

Converting a Date

Let's assume the ENTRYDATE field contains a string date in MM/DD/YY format. We want to convert this string date into a Clarion standard date so we can perform calculations against it and display it in various other formats.

We use the DEFORMAT function to perform the conversion as shown below. The @D1 represents the Clarion Date Picture that corresponds to the MM/DD/YY format. See *Date Pictures* in the *Language Reference* for more information.

```
AssignRecord      ROUTINE
       CLEAR(CUS:Record)
       CUS:NUMBER = IN::NUMBER
       CUS:FIRSTNAME = IN::FIRSTNAME
       CUS:LASTNAME = IN::LASTNAME
       CUS:ADDRESS = IN::ADDRESS
       CUS:CITY = IN::CITY
       CUS:STATE = IN::STATE
       CUS:ZIP = IN::ZIP
       ! CUS:AREACODE =
       CUS:PHONENUMBER = IN::PHONENUMBER
       CUS:ENTRYDATE = DEFORMAT(IN::ENTRYDATE,@D1)
```

How to Make the Transition to the ABC Templates

This FAQ documents development issues that are necessary for Clarion developers familiar with the procedure based *Clarion templates* who are considering the use of the *Clarion ABC* template chain.

You now have a vastly improved development tool at your disposal: **Clarion**, with its Application Builder Class (ABC) Templates and ABC Library. These tools write Object-Oriented (OOP) code for you, using the same template interface that you've grown accustomed to in earlier versions of the Clarion templates. This article provides answers to the most common questions about making the move to ABC/OOP.

So, the first question you'll probably ask yourself (after doing the tutorials in *Getting Started* and *Learning Clarion* to become familiar with the new features) is: "How can I leverage what I already know to get up to speed with Clarion ABC as fast as possible?" That's the overall question that this informal article answers. You'll probably be surprised at how much is still the same, how many resources are at your disposal, and how much more power Clarion has put in your hands with the change to OOP technology.

For those of you who don't write embedded code (or very little) and mostly let the templates handle all the coding, the answer to the question is: you're already leveraged, you already know it--just keep on working with Clarion as you always have. You may notice a reduction in the size of your EXEs and an increase in your applications' performance, but you don't really care how we achieve that, now do you! *This article will be the most benefit to those of you who do a substantial amount of coding for yourselves in embed points.*

I've heard that OOP is really hard to learn.

Like all generalities, this one has a grain of truth at its core. The fact is that, like all new concepts, it will certainly take some work to really "get" the OOP concepts firmly in place, but it really is not as difficult as it's commonly made out to be. It's said, "OOP only takes one minute to understand, but that one minute might not come for a couple of weeks." It just takes a bit of study and a bit of working with it for you to get to the essential "Ah Ha! That's what OOP is all about" bolt of enlightenment.

The biggest difference between writing OOP versus Procedural code is in how you think about the code you write. As one experienced developer is fond of pointing out, in Procedural code you normally think, "How can I perform this task on this thing?" while with OOP code the thought process should be, "I am a thing. How should I perform this task?" These are very different questions representing very different viewpoints.

For example, a Procedural coding thought might be, "How do I make a window refresh when I want it to?" while the corresponding OOP thought should be, "I am a window. How do I refresh myself when necessary?" *You need to change your perspective from an external (doing things to objects) to an internal (each object does its own thing) viewpoint.*

To help you learn the basic theoretical concepts of OOP, SoftVelocity has two articles in the *Clarion Programming Guide* PDF: "Easing into OOP" and "Object Oriented Programming." These two articles both explain the fundamental concepts of OOP theory and how they are implemented in the Clarion Language. The *Language Reference* documentation of the CLASS structure and its related components also covers this same ground in a more formal manner.

OOP, once you've learned the meaning behind all the "buzzwords," is actually pretty simple--it's just not very easy at first. The OOP concepts, while simple, are fundamentally different from Procedural coding concepts. That very difference is what makes OOP seem difficult at first, but it is also that basic difference that gives OOP its power. The result is worth the effort required to attain it.

I've heard that the ABC generated code looks very different.

Yes, that's certainly true. The ABC templates only generate a single line of code (**`GlobalResponse = ThisWindow.Run()`**) into a generated PROCEDURE. All the code to accomplish the functionality of the PROCEDURE is contained either in the ABC Library, or in overridden methods of the ABC Library. These overridden methods are specifically generated for you by the ABC Templates to implement the functionality you request when you fill in the template prompts.

For all the various types of procedures, the ABC template generated code gives you the same functionality the Clarion procedural templates give you (often more, and with more efficient execution). You can easily see this just by looking at the similarity in the template prompts in the Application Generator. *This means that you can simply continue to choose the functionality your PROCEDURE should perform from the ABC Template prompts just as you did before--without changing the familiar way of working you're already accustomed to!*

I've heard that writing embed code is very different in ABC.

This is both true and untrue. True, there are some new things to consider when you write your embed code, but quite often you can write your code just as you did before. In other words, where in the Clarion templates you might write a couple of lines of code into the EVENT:Accepted embed point of a control to set up some variables and call a procedure, or to enable/disable other controls on the same window, or to check the value entered by the user, or … *In ABC you do exactly the same thing--no change at all. For the most part, you can write your event handling embed code in ABC just as you did with the Clarion templates.*

We've added a new feature to the Clarion language specifically to enable you to keep coding your event handling embeds the same way: new scoping rules. In Clarion, PROCEDURE local variables and ROUTINEs are visible and available for use anywhere within that PROCEDURE and any overridden method of a CLASS declared within that PROCEDURE. *This means you can still use local variables and call your ROUTINEs in your embed code exactly as you always have, despite the fact that they are actually in separate procedures.*

So what are the new considerations to bear in mind? Here's the biggest one: **Realize that your embed code is now being placed in a completely separate procedure--an ABC "method" (procedure) specifically overridden by the templates for the individual PROCEDURE you are creating.**

There are a couple of implications to this that might not be obvious:

- Any implicit variables you use are in scope only within that one method. Using the "same" implicit variable in another embed (in another method) now creates a separate instance of that implicit variable unique to that method.

  For example, in the Clarion templates if you assigned a value to X# in EVENT:Accepted for a BUTTON control, then tested the value of X# in EVENT:CloseWindow to see if the BUTTON was pressed, you would be referencing the same X# implicit variable in both statements and your code would work as expected. However, in Clarion ABC generated code, EVENT:Accepted is handled by the ThisWindow.TakeAccepted method and EVENT:CloseWindow is handled by the ThisWindow.TakeWindowEvent method. Since these methods are two separate procedures, you would end up with two different X# implicit variables (one in each method), and your code would definitely NOT work because the value of X# in ThisWindow.TakeWindowEvent would always be zero.

  Therefore, where you once might have used implicit variables across multiple embed points, you should now explicitly declare local variables. *The hidden benefit to this is that by explicitly declaring local variables you are letting the compiler catch any spelling mistakes you might have made in your code (a common problem when using implicit variables).*

- In the Clarion templates you might have placed an OMIT statement in one embed and its terminator in another embed to "work around" template generated functionality that you wanted to override. This is very dangerous to do in the ABC templates, because you'll probably OMIT much more code than you intended to, breaking your application in ways that may not be obvious.

For example, in the Clarion templates if you placed an OMIT in the "EVENT:Accepted -- Before Generated Code" embed point for a control and its terminator in the "EVENT:Accepted – After Generated Code" embed point for the same control, you would simply be omitting template generated code for that one control. However, in Clarion ABC generated code, you would be omitting all the code (generated and embed code) for a number of controls along with the call to the PARENT.TakeAccepted method to handle all the standard EVENT:Accepted functionality.

Therefore, when you want to override any standard Template functionality in ABC, just override the appropriate methontentsod, issuing a RETURN before the call to the PARENT method, and do not use OMIT. *The hidden benefit here is that your ABC generated OOP code is much more flexible and efficient, and you'll probably have less need to override standard template generated functionality.*

What resources do I have to help me learn to convert my code to ABC?

Clarion 6 has an application converter to convert your Clarion template-based applications to use the Clarion ABC Templates. This converter has two modes: Automatic and Manual. *When you operate the converter in Manual mode you have an interactive teaching tool to help you learn all the new ABC ways of coding – and it operates on the code that you wrote!*

Clarion gives you a tool to edit your embedded source: the Embeditor (in the Application Generator, RIGHT-CLICK and select **Source** from the popup menu). The Embeditor was specifically designed to show you all the possible embed points in your PROCEDURE within the context of template generated code. It is important to note here that the Embeditor does not show all the code that WILL be generated for your procedure, but all the code that COULD BE generated. *Seeing your code in context makes it much easier to understand the new scoping issues.*

What's with these new Embed Priorities?

The ABC Templates contain fewer named embed points than the Clarion templates do, so at first it may look like we've removed some capability. However, the new Embed Priorities allow each named embed point up to 10,000 logical embed points within the generated code. This can be pretty confusing until you use the Embeditor to edit your embedded source. Once you can see exactly where the embed point priorities lie in context, you will be able to clearly see where you need to place your code. *The priority numbers themselves do not matter--what matters is where the embed point priority lies within the surrounding generated code, and that's why we gave you the Embeditor.*

The ABC Templates have a new naming convention for the embed points which uses the names of the ABC Library method you're overriding when you place code into the embed point. These may at first seem unfamiliar since they are different than the names used in the Clarion templates. The easy way to learn the new embed points and their priorities in relation to the Clarion template embed points is to:

- Choose **Setup ▸ Application Options** and from the **Action for Legacy Embeds** droplist, select **Show all and generate all** (you can do this while still in your app).
- Use the **Embeds** dialog as you always have to write your embedded source.
- Reset the **Action for Legacy Embeds** to **Show filled and generate all** (to highlight the difference between the embed points by labeling the Legacy embeds as such in the Embeditor).
- Use the Embeditor to edit your source and move your code from the Legacy embeds to the appropriate new embed point (immediately above or below the Legacy embed).

Working this way, you'll find that you're still being productive while you are learning the new ABC embeds, and learning the new ABC Library methods at the same time!

Using this process, you will quickly learn the ABC equivalents for your more commonly used embed points, such as these:

Clarion template embed	**Clarion 6 ABC equivalent**
Initialize the procedure	**WindowManager Method Executable Code Section, Init() Priority: ~6500**
After Opening the Window	**WindowManager Method Executable Code Section, Init() Priority: ~8030**
Beginning of procedure, After Opening Files	**WindowManager Method Executable Code Section, Init() Priority: ~7600**
Preparing to Process the Window	**WindowManager Method Executable Code Section, Ask() Priority: FIRST**
Before Closing the Window	**WindowManager Method Executable Code Section, Kill() Priority: ~7300**
End of procedure, Before Closing Files	**WindowManager Method Executable Code Section, Kill() Priority: ~5600**
Format an element of the Browse Queue	**Browser Method Executable Code Section, SetQueueRecord()Priority: ~5500**
Before Printing detail section	**Process Manager Method Executable Code Section, TakeRecord() Priority: FIRST**

```
Activity for each record (Process template)
                          Process Manager Method Executable Code Section,
                                                  TakeRecord()    Priority: LAST
Browser, Double Click Handler    Browser Method Executable Code Section,
                                                  TakeKey()       Priority: FIRST
             IF RECORDS(SELF.ListQueue) AND KEYCODE() = MouseLeft2
          !Place your code here (you must write the surrounding IF structure, too)
             END

Browser, no records found        Browser Method Executable Code Section,
                                                  ResetQueue(BYTE ResetMode)
                                                    Priority: LAST
             IF NOT RECORDS(SELF.ListQueue)
          !Place your code here (you must write the surrounding IF structure, too)
             END
```

Yeah, but I heard that writing File Handling code is very different in ABC.

OK, here's where there really *is* a fundamental difference. Since the ABC Templates generate their code using the ABC Library, you'll want to write your embedded code to use the ABC Library methods so that there's no possibility that your embedded code will "mess up" anything the template generated code is counting on. Naturally, this is going to be easier to do than you might think.

Here's a table of the ABC methods to use in place of the common Clarion language statements:

Clarion template code	Clarion 6 ABC Library equivalent
`OPEN(File)`	`Relate:File.Open()`
	`!This ensures all related files are opened,`
`SHARE(File)`	`Relate:File.Open()`
	`!  as well as the named file, so Referential`
`CheckOpen(File)`	`Relate:File.Open()`
	`!  Integrity constraints can be enforced.`
`CLOSE(File)`	`Relate:File.Close()`
	`!This ensures all related files are closed.`
`ADD(File)`	`Access:File.Insert()`
	`!These ABC methods all do error handling`
`IF ERRORCODE THEN STOP(ERROR()).`	`!  so the error check is unnecessary. Insert`
	`!  also handles autoinc and data validation.`
`PUT(File)`	`Relate:File.Update()`
	`!The Relate: object enforces RI constraints`
`IF ERRORCODE THEN STOP(ERROR()).`	`!  in Update() and Delete() methods.`
`DELETE(File)`	`Relate:File.Delete(0)`
	`!Parameter suppresses the default confirm`
`IF ERRORCODE THEN STOP(ERROR()).`	`! dialog when 0.`

Another common file handling situation is the simple file processing LOOP. In the Clarion template embeds, you would write code like this:

```
SET(key,key)
LOOP
NEXT(File)
IF ERRORCODE() THEN BREAK.              !Break at end of file
!Check range limits here
!Process the record here
END
```

And here is the equivalent ABC code:

```
SaveState = Access:File.SaveFile()
!Tell ABC to "bookmark" where it's at (just in case)
SET(key,key)  !Note there's no change here
LOOP
 IF Access:File.Next()<>Level:Benign THEN BREAK.
 !Breaks when it tries to read past end of file
 !Check range limits here
 !Process the record here
END
Access:File.RestoreFile(SaveState)
 !Undo the "bookmark" (SaveState must be a USHORT)
```

As you can see, this is all pretty straightforward--only a couple of minor changes to learn.

Another common code construct is getting a record from a file. In the Clarion template embeds, you might write code like this:

```
IF File::used = 0
   CheckOpen(File)
END
File::used += 1
CLEAR(FIL:record)
Fil:Code = 123
GET(File,FIL:CodeKey)
IF ERRORCODE() THEN CLEAR(FIL:Record).
File::Used -= 1
IF File::used = 0
   CLOSE(file)
END
```

And here is the equivalent ABC code:

```
Relate:File.Open()            !This handles all error conditions
CLEAR(FIL:record)
FIL:Code = 123
Access:File.Fetch(FIL:CodeKey)!Fetch clears the record on errors
Relate:File.Close()
```

And of course, the file Open and Close method calls can be generated for you if you just add the file to the procedure's File Schematic. The ABC Library is smart enough to only open a file if it really needs to, making your program more efficient. *Using Clarion's ABC Library methods you write less code to accomplish the same (or more) functionality.*

How do I learn about all these new ABC methods?

The easiest way to become familiar with the ABC Library overall is to read the *ABC Library Reference* --specifically those sections which present an Overview of each Class. It's not necessary (at first) to read about every single method, just use the Overview to get an idea of what each Class is all about and what you can do with them. Then read the descriptions of the methods that look like they'll be immediately useful to you.

Method names have specific meaning in the ABC Library to indicate the type of functionality each method provides. All the classes consistently use these names:

Add*Item*	Adds an *item* to its object's datastore. The *item* may be a field, a key, a sort order, a range limit, another object, etc.--anything the object needs to do its job.
Ask[*Information*]	Interacts with the end user to get the *Information*.
Fetch	Retrieves data (usually from a file).
Get*Item*	Returns the value of the named *item*.
Init	Does whatever is required to initialize the object.
Kill	Does whatever is required to shut down the object, including freeing any memory allocated during its lifetime.
Reset[*what* or *how*]	Resets the object and its controls. This includes reloading data, resetting sort orders, redrawing window controls, etc.
Set*Item*	Sets the value of the named *item*, or makes the named *item* active so that other object methods can operate on the active *item*.
Take*Item*	"Takes" the *item* from another method or object and continues processing it. The *item* may be a window event (Accepted, Rejected, OpenWindow, CloseWindow, Resize, etc.), a record, an error condition, etc.
Throw[*Item*]	"Throws" the *item* to another object or method for handling. The *item* is usually an error condition.

Try*Action*	Makes one attempt to carry out the *Action*. A return value of zero (0 or Level:Benign) indicates success; any other value indicates failure.

Knowing these consistent naming conventions will make it much easier to understand what an object's methods do, whether you've read the *ABC Library Reference* about that specific type of object or not!

In addition to the *ABC Library Reference*, Clarion also has a Class Viewer to show you the ABC Library properties and methods in a tree view. On any Classes tab, just press the button labeled "Application Builder Class Viewer" to view the ABC Library structure. *The Class Viewer graphically shows you how the ABC Library classes are derived--which class inherits properties and methods from which Parent class.*

The ABC Template set contains two Code Templates, which will help you learn more about using the ABC Library: *CallABCMethod* and *SetABCProperty*. These were specifically created to "walk you through" writing ABC Library code in any executable code embed point. *These two Code Templates will write your method calls and object property assignments for you!*

So how do I figure out which ABC method to use?

Here is a standard process you can use to accomplish any task using the ABC Templates and Library:

1) Determine if the ABC Templates can perform the task.

The ABC Templates do many things for you that the Clarion templates do not. See the ReadMe file for a list of the new template features (the *Template Guide* contains complete ABC Template documentation).

If the ABC Templates will perform the task for you, you're done. If they won't, continue on to Step 2.

2) Identify the ABC Object (and its CLASS) that manages the behavior you need to change.

Quite often there are several candidate objects in a single PROCEDURE that might control the behavior you need to change. Therefore, you may need to identify several objects/classes to look at in the following steps. The easiest way to identify candidate objects is to look in the Embeditor. Each object you see in your PROCEDURE is derived from the ABC Library. The object's Parent CLASS name (in parentheses following "CLASS") should give you an idea of the type of behavior the object controls. The most common objects and their corresponding classes are:

Object	Parent CLASS
ThisWindow	WindowManager
BRW*n*	BrowseClass
ThisProcess	ProcessClass
ThisWindow	ReportManager (in Report procedures)
ThisReport	ProcessClass (in Report procedures)

Once you've identified the candidate objects and CLASSes, continue on to Step 3.

3) Determine if the object/CLASS has a property that you can set to accomplish the task.

The *ABC Library Reference* and the Class Viewer are the best sources for this information. Remember that a derived class inherits all the properties of its parent class.

If setting a property performs the task for you, you're done. If not, continue on to Step 4.

4) Determine if a method of the object/CLASS already does the task.

Again, the *ABC Library Reference* and the Class Viewer are the best sources for this information. Remember that a derived class inherits all the methods of its parent class.

If calling an existing method performs the task for you, you're done. If no method already does the task, continue on to Step 5.

5) Determine which method you can modify to accomplish the task, then override it.

Follow the same process as in Step 4 to identify the method that comes closest to performing the task you need to accomplish. To override a method, place your code in the embed points for that method. You can use other ABC properties and methods in your code to do what you need.

If the modified/overridden method is VIRTUAL, you're done. You do not need to call VIRTUAL methods yourself since the other methods of the CLASS call them for you as part of their normal operations (as in the above example). If the method is not VIRTUAL or you need to call it outside the normal sequence of events, then you just use a similar process to determine where to call the method: find the method that manages the behavior and embed your call in an embed point of that method.

Following these steps, you can accomplish any programming task in the most efficient manner, by always working at the highest level of abstraction possible in Clarion--that is, do as little "work" yourself as possible to accomplish the greatest effect.

OK, the theory looks good, but how about some real examples?

Sure. Here're a few:

Task: I want a Browse list to display in descending key order.

1) The Browsebox Control Template's documentation in the *Template Guide* tells you that it supports extra sort fields in addition to key fields. If you do not specify a key in the File Schematic, the Browse will sort just by the fields you name as Additional Sort Fields. Therefore, just list the key fields in Additional Sort Fields with a leading minus sign and a comma between each field (such as: -CUS:LastName,-CUS:FirstName). *You're done at Step One!*

Task: I want to print a report, skipping the print preview if the user has elected not to preview.

1) The Report Template's documentation in the *Template Guide* shows no template settings to conditionally toggle preview (it's either always on or always off), so you go on to Step Two.

2) Opening the Embeditor for the report, you see two objects that appear to be likely candidates to control this behavior: Previewer/PrintPreviewClass and ThisWindow/ReportManager. PrintPreviewClass is pretty obvious. ReportManager is also likely because you can see it has a method called AskPreview (so it might have something to do with the preview functionality). You go on to Step Three.

3) The *ABC Library Reference* documents all the public properties for each class. Checking the properties for PrintPreviewClass, you see that there is no property that appears to control whether the print preview executes or not. However, when you check the properties for ReportManager you see a SkipPreview property that does exactly what you need. Therefore, to conditionally suppress print preview based on a user setting in a control file you can simply:

```
IF NOT CTL:PrintPreview THEN ThisWindow.SkipPreview = TRUE.
```

Where do you put this one line of code? Since this property is in the ThisWindow object it cannot execute until the ThisWindow object has been initialized (following the PARENT.Init call in ThisWindow.Init). And, since the purpose of this task is to "shortstop" the Print Preview, it must execute before the Previewer object is called to display the finished report (before the PARENT.AskPreview call in ThisWindow.AskPreview). These two requirements limit you to probably only a few hundred embed points within your report procedure's logic, but the best place to just "set it and forget it" is the end of the ThisWindow.Init method. *You're done at Step Three!*

Task: I want to let the end-user dynamically filter a browse list at runtime.

1) The BrowseBox Control Template's documentation in the *Template Guide* doesn't show any template settings for a dynamic filter, so you go on to Step Two.

2) In the Embeditor, you see that the only likely candidate object/CLASS is BRW1/BrowseClass, because it controls the browse list, then go on to Step Three.

3) None of the BrowseClass properties listed in the *ABC Library Reference* appear to help for this task. BrowseClass is derived from the ViewManager, which means it inherits all the ViewManager properties and methods, so you check the ViewManager properties also. None of them apply either, so you go on to Step Four.

4) The *ABC Library Reference* documents the BrowseClass methods in addition to its properties. You don't immediately see any that might apply, so you check the inherited ViewManager methods, also. That's where you find the ViewManager's SetFilter method, which appears to be exactly what you need for this task. Therefore, you see that to change the browse filter you can execute code like this:

```
IF NewFilter !If the user entered a letter
   BRW1.SetFilter('CUS:LastName[1] = ''' & NewFilter & '''')
ELSE !Limit to names starting with the 1 letter in NewFilter
   BRW1.SetFilter('') !Reset to no filter
END
ThisWindow.Reset(1) !Force-Refresh the window display
```

But where to put this code? Since you want the end user to enter the data to filter on, you obviously need an ENTRY control the user can type into, and that ENTRY control's USE variable is obviously the NewFilter variable (a STRING(1) in this example code) you want to use in the SetFilter method. Therefore, the only logical choice for where to place this code is in the EVENT:Accepted embed point for the NewFilter ENTRY control.*You're done at Step Four!*

Task: I want the user to be able to copy an existing record when adding a new record.

This task actually means that you want to provide a "Copy" button that does everything the "Insert" button does (like auto-increment) but gives the user some "starting point" data in the update Form based on the currently highlighted record in the Browse list.

1) Neither the BrowseBox nor the BrowseUpdateButtons documentation in the *Template Guide* shows any template settings for this, so you go on to Step Two.

2) In the Embeditor, you see that the only likely candidate object/CLASS is BRW1/BrowseClass, because it controls the browse list and calling the update Form, so you go on to Step Three.

3) None of the BrowseClass or inherited ViewManager properties listed in the *ABC Library Reference* appear to help for this task, so you go on to Step Four.

4) You look at the BrowseClass methods and don't see any that might apply, so you check the inherited ViewManager methods, also. That's where you find the ViewManager's PrimeRecord method, which takes a parameter that appears to be exactly what you need for this task. However, this method is VIRTUAL--automatically called by the base classes--so you don't need to call it, so you go on to Step Five.

5) Opening the Embeditor for the Browse procedure, you find the BRW1.PrimeRecord method. You can see that it takes a parameter called SuppressClear which defaults to FALSE (zero). Therefore, you know that you can just assign TRUE (one) to the SuppressClear parameter before it is passed on to PARENT.PrimeRecord to suppress clearing the rest of the fields in the record (after auto-incrementing any necessary fields). However, if you just do a simple assignment statement, you will be overriding this functionality for every insert. Therefore, you need to declare a local flag variable to the Browse procedure to flag whether or not to suppress the clear. Scroll back up to the top of the file and, in any Data Section embed point, add the following code:

```
CopyFlag BYTE                              !Declare the flag variable
```

Then you can place the following code in the BRW1.PrimeRecord method immediately BEFORE the PARENT.PrimeRecord call:

```
IF CopyFlag = TRUE          !Check the flag's value
   SuppressClear = TRUE    !Set the PrimeRecord parameter value
   CopyFlag = FALSE        !Reset the flag
END
```

This overrides the ABC Library's PrimeRecord method with your own, but still calls the ABC Library's method to handle its functionality (so we don't have to re-write its code). Since this is a VIRTUAL method, you don't have to worry about calling it--it's called automatically for you from within the other ABC Library methods.

You're almost done, but not quite. In order for the user to signal when they want to copy a record, you need to add a "Copy" BUTTON control to the Browse window. Then, in EVENT:Accepted for your ?Copy BUTTON, you place the following code to set the CopyFlag and initiate adding the copied record:

```
CopyFlag = TRUE                   !Set the flag and then trigger
POST(EVENT:Accepted,?Insert)       ! the Insert button's code to
execute as normal
```

When the user presses your Copy button, CopyFlag is set to TRUE and then the normal Insert button code sequence occurs, but your overridden BRW1.PrimeRecord method will execute instead of the ABC Library's standard method. *You're done at Step Five!*

Task: I want to add my new "Copy" button to the Browse's popup menu.

This task requires that you have a "Copy" button as described above.

1) The BrowseBox Control Template documentation in the *Template Guide* doesn't show any template settings for this, so you go on to Step Two.

2) In the Embeditor, you see that the only likely candidate object/CLASS is BRW1/BrowseClass, because it controls the browse list, then go on to Step Three.

3) The BrowseClass documentation in the *ABC Library Reference* lists a Popup property described as a "browse popup menu reference." Reading the description of this property you discover that it is a reference to the PopupClass object used by the Browse, so you look at the PopupClass to see if there are any properties that will add an item to the popup menu for you. There are no PopupClass properties, so you go on to Step Four.

4) You look at the BrowseClass methods and don't see any that might apply, so you check the PopupClass methods, also, since you know the Browse object contains a reference to the PopupClass object. That's where you find the PopupClass's AddItemMimic method, which will add an item to the popup menu to press a BUTTON control. This appears to be exactly what you need for this task. Therefore, you now know that to add the Copy button to the popup menu for this BrowseBox, you can simply execute:

```
BRW1.Popup.AddItemMimic('Copy',?Copy)
!Call a PopupClass method through the
! Browse object's PopupClass reference property
```

So where does this one line of code need to go? Since the Popup property must already exist, it must come sometime after the BRW1 object has been initialized. And, since you just want to add this to the popup menu and not dynamically enable/disable it, this code needs to happen on the way into the procedure, before the user can do anything on the window. In the Embeditor, you can see that the BRW1.Init method is called in the ThisWindow.Init method, and you already know that ThisWindow.Init is always the first method called in any procedure with a window. Therefore, the best embed point to place this one line of code would be one near the end of the ThisWindow.Init method. *You're done at Step Four!*

Task: I want to call a Form directly from the menu (without a Browse) to add records.

This task actually implies the need to do three things: 1) alert the procedure that it will be inserting new records into the database, 2) ensure that any auto-increment keys are properly incremented, and 3) ensure that all other fields in the file are properly initialized to their default initial values.

1) The Form Template's documentation in the *Templates By Topic Guide* doesn't show any template settings for this, so you go on to Step Two.

2) In the Embeditor, you see the only object in the Form procedure (other than two Toolbar-related objects) is ThisWindow/WindowManager. However, you also have a global object for each of the data files your application accesses. Therefore, a second candidate object/CLASS to look at for this task is Access:FileName/FileManager.

   While exploring in the Embeditor you notice that the very beginning of ThisWindow.Init is where the value of the GlobalRequest variable (which tells the Form procedure what file action to perform) is obtained. You know you need to deal with this issue, so in the very first embed point available in ThisWindow.Init (immediately following the CODE statement) you add:

```
GlobalRequest = InsertRecord !Set Form procedure to insert records mode
```

   Having dealt with the first issue, you go on to Step Three.

3) None of the WindowManager properties listed in the *ABC Library Reference* appear to help for this task, so you check the FileManager properties also. None of these apply either, so you go on to Step Four.

4) The *ABC Library Reference* shows two FileManager methods (PrimeRecord and PrimeAutoInc) which appear to be exactly what you need for this task. Now you just need to find where to call them, so you go on to Step Five.

5) The *ABC Library Reference* shows a WindowManager method called PrimeFields that looks like the best place from which to call the FileManager methods. In the Embeditor, you write code like this into any embed point in ThisWindow.PrimeFields:

```
Access:FileName.PrimeAutoInc()    !Auto-increment key fields
Access:FileName.PrimeRecord(TRUE)!Init other fields (suppressing clear)
```

   Now, when you call this Form procedure directly from a menu (without an intermediate Browse) it will automatically be in Insert mode and will correctly handle auto-incrementing keys and setting up all the initial values you declared in the Data Dictionary. *You're done at Step Five!*

Task: I want to allow the end user to pause/resume report generation.

1) The Clarion ABC Templates have a "Pause Button" Control Template which does exactly this. Simply place this Control Template on the Report's ProgessWindow, *and you're done at Step One!*

Task: I want total fields on one tab and the BrowseBox which they total on another.

This task implies that the tab which the user sees first is a "summary" tab, and the "detail" Browse list (or multiple lists) which calculates the totals is hidden on another tab. Simply placing the total fields on a separate tab from the hidden Browse means that the total fields display zeroes until the user has gone to the tab containing the Browse list and come back to the tab containing the total fields. The ABC Templates default to only updating a Browse list when it is visible because this achieves better overall performance for the majority of cases. However, in this one case, the Browse list needs to be active despite the fact that it is not visible to the user.

1) The BrowseBox Control Template documentation in the *Template Guide* doesn't show any template settings for this, so you go on to Step Two.

2) In the Embeditor, you see that the only likely candidate object/CLASS is BRW1/BrowseClass, because it controls the browse list, then go on to Step Three.

3) The BrowseClass documentation in the *ABC Library Reference* lists an ActiveInvisible property which, when set to one (1), makes the Browse list active even though it's not visible. Therefore, you now know that you can simply execute:

```
BRW1.ActiveInvisible = TRUE              !Set the browse to always active
```

So where does this one line of code need to go? Since the ActiveInvisible property must already exist, it must come sometime after the BRW1 object has been initialized. And, this property needs to be set on the way into the procedure, before the user can see the window so the totals will be calculated and displayed correctly. In the Embeditor, you can see that the BRW1.Init method is called in the ThisWindow.Init method, and you already know that ThisWindow.Init is always the first method called in any procedure with a window. Therefore, the best embed point to place this one line of code would be near the end of the ThisWindow.Init method. *You're done at Step Three!*

As you can see, this 5-step process leads you to the earliest possible solution to any programming task.

Are there any other Clarion OOP Learning Resources?

Yes, there are several:

- Clarion Magazine is located at http://www.clarionmag.com (this requires a subscription) and is published monthly by CoveComm Inc. This "e-zine" features articles of interest to all Clarion developers, including writing OOP code and using the ABC Templates and Library.

- Many Clarion developers are on-line and willing to help others as needed. Your colleagues can be found in the SoftVelocity Internet newsgroups--most notably, *comp.lang.clarion* (on many news servers) and softvelocity.*products.C60EE* (only on the *news.softvelocity.com* news server).

Between these external resources, the information presented in this article, and all the Clarion documentation (both printed and electronic), you have a tremendous amount of information to help you learn how to use Clarion's ABC Templates and Library. Keep studying and working with it and you ***will*** get to that essential "Ah Ha! That's what ABC/OOP is all about" bolt of enlightenment--probably sooner than you might think.

How to Manage Threads

This topic will show you how to limit an MDI Child browse procedure to a single instance using messaging.

The simplest solution is to disable the menu item when the browse procedure is active. You can't do this in the menu itself--you must send a message to the Frame procedure from the browse.

First you need to declare two new events in the Global Properties, Global Data embed point:

```
EVENT:DisableCustomerItem     EQUATE(401h)
EVENT:EnableCustomerItem      EQUATE(402h)
```

(Note that user-defined events must start after 400h.)

You also need a global variable, define this in **Global Properties, Data.** Call this **GLO:MainThreadNo**, make it a BYTE.

In the Frame procedure, *WindowManager Executable Code--Init* embed, type the following:

```
GLO:MainThreadNo = THREAD()
```

(Whenever an MDI procedure is STARTed, a thread number is allocated to it. We need the thread number in order to post a message to the frame procedure.)

Now, still in the frame procedure, you need to write code in the *WindowManager Executable Code--TakeWindowEvent [Priority: 2800]* embed to handle the two user-defined events that the frame procedure will receive.

```
  OF EVENT:DisableCustomerItem
    DISABLE(?ShowCust)
  OF EVENT:EnableCustomerItem
    ENABLE(?ShowCust)
```

In the Browse procedure, in the *WindowManager Executable Code--Init [Priority:5600]* embed, type:

```
  POST(EVENT:DisableCustomerItem,,GLO:MainThreadNo)
```

In the *WindowManager Executable Code--Kill [Priority:8000]* embed, type:

```
  POST(EVENT:EnableCustomerItem,,GLO:MainThreadNo)
```

Now, the first time you select the Browse Procedure from your menu, ShowCust starts up, the POST() statement executes and an **EVENT:DisableCustomerItem** is sent to the Frame procedure.

If the user then clicks on the menu again, the message is processed and the item is disabled.

As the user exits ShowCust, the **EVENT:EnableCustomerItem** message is sent to the Frame. When that message is processed the menu item is enabled again.

Why store the Frame's thread number - surely it would always be number 1? Well it might NOT be the first thread in the application.

Thanks to Rob Mousley of Chariot Software for submitting this topic.

How to Minimize a Window

A Minimize button is added to a WINDOW if you specify an ICON for the WINDOW. When the user presses the Minimize button, the window is reduced to an Icon.

In the Window Formatter:

1. Make sure the Window is selected.

2. Choose **Edit ▸ Properties** (or press ENTER).

The **Window Properties** dialog appears**.**

3. Select the **Extra** tab.

4. In the **Icon** combo box, select a standard icon, type the name of the icon file, or select Select File... to locate an icon file using the standard Open File dialog.

5. Press the **Ok** button to close the **Window Properties** dialog.

How the Print Engine Processes Report Sections at Runtime

Before learning how to create a report using the Report Formatter, it's important to understand how Clarion executes a report--in other words, the division of labor between the print engine and your source code, and the order in which Clarion processes all the sections of your report. Each section of the report is a data structure, and each in turn is contained in the REPORT structure.

Smart Processing

The REPORT data structure contains all the information necessary for formatting and printing each page. It automatically handles page overflow management, including widow and orphan control. This frees you from worrying about the "mechanics."

This means that the Clarion executable code to print a report is simple and clean. The following example shows how a total of six lines of executable code can access the file and print a fully-formatted listing of all Customers. Since the Report Formatter writes the entire REPORT data structure, this is all the code the programmer has to write:

```
CODE
OPEN(CustReport)          !Open report for processing
SET(Cus:NameKey)          !Top of file, alpha order
LOOP                      !Process the entire file
 NEXT(CustomerFile)       !one record at a time
  IF ERRORCODE() THEN BREAK.  !Check for errors
 PRINT(Rpt:Detail)        !PRINT tells the REPORT structure to do the work.
END
```

Of course, if you're using the Application Generator, you don't even have to write that much! In the example above, the PRINT statement prints a DETAIL structure for each record in the file retrieved with the NEXT statement inside the LOOP .

The REPORT data structure contains the items that belong on the page, plus the attributes that determine how they appear there. Since you visually design these in the Report Formatter, the code example above really is all you have to do to print the report.

The PRINT statement automatically initiates the page overflow handling. This means that when the LOOP goes through enough records to fill up the page, it automatically generates any other structures on the page--the FOOTER, for example. Then it sends the entire page to the print spooler.

Order

The parts are wholly contained and managed within the REPORT data structure. The parts of the data structure are the FORM, PAGE, HEADER, DETAIL, and page FOOTER; their functions are fully described below. The REPORT data structure may also contain group BREAK structures. Each group BREAK structure can contain its own group HEADER, DETAIL, and FOOTER.

Normally, you want to design reports with only one DETAIL. When you generate reports using the Application Generator, they will probably have only one. This usually is the one inside the group BREAK structure. The Report Formatter adds a DETAIL for each BREAK, for flexibility. You can delete the ones not used.

Once you know the order in which the parts generate at print time, you can understand how to use them better. For the following example, assume a report utilizing all the parts listed above, containing one group structure, with a DETAIL inside. Immediately upon the PRINT command:

1. The print engine composes the FORM, but does not send it to the print spooler yet. The FORM generates only once; the application repeats the FORM and does not recompose it for additional pages.

2. The print engine composes the page HEADER.

3. The print engine processes the group HEADER.

4. The application generates the DETAIL section (within the BREAK) for as many times as will fill the first page, continuously checking for group BREAKs.

If a BREAK occurs on the page:

5. The print engine composes the group FOOTER for the first group.

6. The print engine composes the group HEADER for the next group.

7. The application generates the DETAIL section for the next group of records, continuously checking for further group BREAKs.

At the bottom of the page:

8. The print engine checks for widows, increments the page count, and checks the next page for orphans.

9. The print engine composes the page FOOTER.

10. The print engine sends the entire first page to the print spooler.

11. For page two, since the FORM was composed already, it does not get regenerated, though it will print on the page. The application proceeds directly to the page HEADER.

12. The application repeats the procedures above for this and any additional pages.

Flexibility

The page-oriented nature of the Report Formatter is the key to its flexibility. The print engine composes each page in its entirety before sending it to the printer. This means you may arrange the parts of the report into any page layout you wish.

You can place the FORM, page HEADER, and page FOOTER structures anywhere on the page, within certain limitations. Their placement does not affect the order that the application generates the parts.

That means you can physically place a page FOOTER above a page HEADER. Since the FOOTER generates only after the report processes all the records on the page, this allows you, for example, to place a page total above the records on the page.

You set the position and size of the DETAIL structure as an offset vs. the last DETAIL printed. The print engine prints each DETAIL from page top to page bottom. If the DETAIL is narrow enough so that more than one fits across the width of the page, they print in order left to right, top to bottom.

Group BREAK structures--group HEADER, group DETAIL and group FOOTER--all print as offsets within the DETAIL area, one after the other.

You can do some fancy footwork in cooperation with the print engine. For example, because the DETAIL structure must be printed with the PRINT statement, you can utilize embedded source to place conditional statements within your executable code, to print one DETAIL upon one condition, and another upon a different condition.

As long as you remember the order in which the application generates each section, which determines the current record and the values of the totals, tallies and other operations on the fields in each structure, you can build in a great deal of flexibility within the REPORT data structure, and let the print engine worry about fitting it all onto the page at runtime.

How to Print Grand Totals on a Report

To print Grand Totals on a report:

1. Add a new Detail band outside of any BREAK structures you have in the report, then follow the steps in How to suppress printing a Detail Band until explicitly called to print to always suppress printing it.

2. Place your grand total fields on the new Detail band. Be sure to set **Reset on** to <None> and specify the detail or break structures on which to tally the total in the **Tallies** list on the **Extra** tab.

3. Explicitly print your grand total detail band in the Before Closing Report embed point:

```
PRINT(RPT:MyGrandTotalDetailBand)
```

How to Print Labels

Note:

Before reading this FAQ, see a new and easier way to design labels using the Label Wizard in the Template User's Guide PDF. Printing labels simply means printing a multi-column report, that is, getting the report rows and columns to match up with the commercial label forms you use.

1. With a ruler, measure the height and width of the label paper, measure the height and width of the individual labels, and measure any top or left margins on your label paper. Make your measurements in inches.
2. Create a report of your address file. Use the Report Wizard if you want, but don't worry about formatting yet. Just make sure the report contains all the address fields you need for your labels.
3. From the Application Tree, RIGHT-CLICK your report procedure and choose Report from the popup menu.
4. Delete all report sections, except the Detail section.
5. Choose **Edit ▸ Report Properties**.
6. On the **General** tab, choose 1/1000 inches in the **Units** drop down list.
7. On the **Position** tab, set the margins you measured earlier. In the **Top Left Corner** group, the **X** field represents the left margin in thousandths of and the **Y** field represents the top margin in thousandths of inches. So, if the left margin of your label paper is 1/2 inch, type 500 in the **X** field. If the margin is zero, type zero in the **X** field. Do the same for the top margin and the **Y** field.
8. On the **Position** tab, set the height and width of the label paper. In the **Width** group, click on fixed, and type the paper width in thousandths of inches. Standard letter size paper is 8 1/2 inches, so type 8500. Do the same for the paper height. Standard letter size paper is 11 inches, so type 11000.
9. Press the **OK** button to return to the **Report Formatter**.
10. Arrange your address fields in a vertical format, that is, one field below another near the left margin. Use the Alignment tools for precise alignment.
11. RIGHT CLICK the detail band and choose Position from the popup menu.
12. On the **Position** tab, set the height and width of the individual labels. In the **Width** group, click on fixed, and type the label width in thousandths of inches (e.g., for a 2 1/2 inch label, type 2500). Do the same for the label height.
13. Press the **OK** button to return to the **Report Formatter**.
14. Readjust the position, size, and font of your address fields if necessary.
15. **Preview**! your label report.
16. **Exit**! the **Report Formatter** and save your changes.

How to Print One Record per Page

To print a separate page for each record, check the PAGEAFTER box in the **Detail Properties** dialog.

How to Print to a File

You can change the windows default printer without calling the PRINTERDIALOG function. This can be done by using Clarion's property syntax. The property to use is **PROPPRINT:Device**. This property definition can be found in the ..\LIBSRC\PRNPROP.CLW. This must be included in your application before making use of any of the properties defined therein.

Install the Generic Printer Driver

1. In Windows 3.x, use Control Panel. In Windows 95, choose Settings ▸ Printers from the Start Menu, then double-click on the Add Printer icon.

To include PRNPROP.CLW:

1. Press the **Global** button on the Application Tree, to open the **Global Properties** dialog.
2. Press the **Embeds** button.
3. Select the *After Global Includes* embed point and add the following embedded source code:

```
include('prnprop.clw')
```

To change the printer device:

1. From the Application Tree, select your report procedure and press the **Properties** button.
2. Press the **Data** button. Create a STRING variable named: sav::printer. Close the Local Data dialog.
3. Press the **Embeds** button.
4. Select the *ProcessManager Method Executable Code--Open (Priority:2500)* embed point and add the following embedded source code:

```
sav::printer = PRINTER{PROPPRINT:Device}      !  save windows default printer
PRINTER{PROPPRINT:Device} = 'Generic / Text Only'! set to ASCII device
PRINTER{PROPPRINT:PrintToFile} = TRUE         ! print to file flag
PRINTER{PROPPRINT:PrintTo Name} = 'REPORT.TXT'! set filename for report
```

At the end of the report, restore the original default printer:

1. From the Application Tree, select your report procedure and press the **Properties** button.
2. Press the **Embeds** button.
3. Select the ProcessManager Method Executable Code--Close (Priority:7500) embed point and add the following embedded source code:

```
PRINTER{PROPPRINT:Device} = sav::printer
PRINTER{PROPPRINT:PrintToFile} = FALSE
! print to file flag
```

How To Put Your Program To Sleep

There are some occasions when you may want to introduce a timed delay to your Windows applications. You don't want to use a LOOP structure to simulate a delay because the LOOP command is processor dependant. Looping 1000 times on one machine may be quite different from another in regards to length of time or duration.

Instead, use the Windows Sleep API function, which releases the CPU and lets other apps in the system effectively multitask. The Sleep function suspends the execution of the current thread for a specified interval.

To add the SLEEP prototype to your application, include the following MODULE structure inside the Global Map of your program or application:

```
MODULE('')
 SLEEP(LONG),PASCAL
END
```

The SLEEP API has a time parameter, in milliseconds, for which to suspend execution. A value of zero causes the thread to relinquish the remainder of its time slice to any other thread of equal priority that is ready to run. If there are no other threads of equal priority ready to run, the function returns immediately, and the thread continues execution.

Besides using SLEEP() to introduce a delay, at times it is also useful to allow other programs to "catch up" with your applications, and SLEEP() can be useful for this. For example, consider the following sample code:

```
SETCURSOR(Cursor:Wait)
System{Prop:DDETimeout} = 12000
WordChannel = DDEClient('WinWord','System')
IF NOT WordChannel
   RUN('C:\Program Files\Microsoft Office\Office\WinWord.exe',0)
   SLEEP(50) !Introduce a little delay to allow MSWord to initialize.
END

WordChannel = DDEClient('WinWord','System')


DDEExecute(WordChannel,'[Insert "My text" + CHR$(10)]')
DDEClose(WordChannel)
SetCursor(Cursor:Arrow)
```

On slower machines, it is possible that the MSWord program may not finish its initialization sequence on start up before the DDEClient command is executed. SLEEP() allows the MSWord program to finish its startup and open a proper DDE channel to the Clarion program.

How to Register a Template Set

You can register template sets shipped with this product, your own template sets, or third-party vendors' template sets.

1. Choose **Setup ▸ Template Registry.**

   The Template Registry lists all registered templates.

2. Press the **Register** button.

   Select the template (.TPL) file(s) from the standard Open File dialog.

3. Press the **OK** button.

How to Repair Data Corruption in TopSpeed/Clarion Files

The TopSpeed file system offers high performance, high security, and excellent compression for the Clarion development environment. Concerns have been raised about the stability of the driver, because of error messages - posted from the driver - when applications are running and accessing TopSpeed files. TopSpeed files are not more susceptible to corruption than other files; however, error detection is higher because the driver engine is extremely sensitive to the smallest anomaly in the data file.

Performance and security standards require the driver to have very aggressive error checking algorithms. The integrity of the data written into the file is a major concern. For this reason, writes to the data file are handled in two steps. First the data record is written to the file and a flush is issued to clear the buffer and write to the physical disk. Only then is the header of the file updated to reflect the new record and a flush is issued again.

The network hardware and software is responsible for physically writing the information to the disk. If a bad bit is transferred and then written into the file, it is usually identified, by the driver, when the record is decompressed. During decompression, the actual record size is compared to the size reported in the header. If either of these was written incorrectly to the file, the two sizes will not match. This detection will cause the driver to report an error or corruption in the file (tpsbt.cpp line XXXX).

A variety of data files have been tested extensively for corruption, and at this time only one problem has been identified with the file driver. This data corruption occurs if the following code is issued:

```
STREAM(file)
LOGOUT(1,file)
ROLLBACK
```

This corruption can be prevented with the following change to the code:

```
STREAM(file)
FLUSH(file)
LOGOUT(1,file)
ROLLBACK
```

Excluding the above problem, all known bugs within the TopSpeed driver have been fixed with the release of Clarion for Windows Version 2.003.

Clarion data files, while not as stringent on error checking as TopSpeed files, will post errors if a bad bit is written into a field containing key information. If the record and key are not matching, then the driver will post and Invalid Key File error (Error 38) or a Key File Be Rebuilt Error (Error 46).

Both TopSpeed and Clarion files operate in a multi-user environment through shared data access in a station-to-station manner. This means that there is no particular station that works as a server controlling the requests made to the files. This type of data environment requires that the underlying operating system and network requesters work flawlessly to ensure proper passing of data from station to station. If this layer is faulty or unstable, incorrect bits could be sent to the data files causing data corruption.

If consistent file corruptions are occurring on applications using TopSpeed or Clarion data files, there are two steps to fixing these problems. First, fix the corrupted data file; second, prevent further corruption.

1. Fix the corrupted data file.

TopSpeed data files: TPSFix will recover most corrupted TopSpeed files. Make sure that the box for Build Keys is checked so that the utility will build all the keys. In some corruption cases, TPSFix will initially report no errors found in the file. If this is the case, continue to run the Fix utility to the end of the file. It may still find errors when reading through the file. See Chapter 33 in the Programmer's Guide for more information on the TPSFix utility.

If TPSFix is unable to correct the file, a conversion program can be created which will loop through the file in record order and rebuild the keys. Refer to the Users Guide, Chapter 17, Generating Source for File Conversion, for a step by step explanation for creating a simple conversion program. The Source Structure and Target Structure will be the same in this case, you may wish to write the new file out to a new name.

It may be necessary to only rebuild the key files for the .tps file, for the corruption to be fixed. This can be done in a application by issuing the BUILD(file) statement after the file has been OPENed. Before issuing the BUILD() statement, make sure to issue to the FULLBUILD=ON command to the file driver. This insures that all the keys are rebuilt, even if no changes have been made. FULLBUILD is usally set to OFF to optimize the speed of the driver, so it should be set to ON before the BUILD and OFF after the BUILD is completed, such as:

```
OPEN(file)
SEND(file,'FULLBUILD=ON')
BUILD(file)
SEND(file,'FULLBUILD=OFF')
```

Clarion data files: Because the problem is generally a corrupted key, the key can be deleted and then rebuilt. Rebuilding the key can be done within the Database Manager (CW), by selecting the file and then pressing the Browse button. When prompted to rebuild the keys choose yes. This should fix the corrupted key.

If the automatic rebuild does not work, use the conversion program method described for the TopSpeed files above.

2. Prevent Further Corruption

Once the file has been corrected and replaced on the network, the underlying reason for the error needs to found and fixed. At this time, three specific circumstances have been identified as causing corruption to TopSpeed and Clarion data files.

1. **Virtual Network Redirecter with Windows 95**. The version of the redirecter that ships with Windows 95 (prior to last quarter 1996) has buffering problems when retrieving data from a Windows NT server. The directer would automatically set local caching, which would cause the corruptions.

Other vendors have experienced similar file corruption and Microsoft has offered a fix. A copy of the new file can be downloaded from the Data Flex CompuServe Forum, library 1, filename VRDR2.EXE , or from the following FTP site -- ftp.dataaccess.com/pub/redir, the filename is vrdr2upd.exe. Run the executable and the new redirector will install itself and make changes to the registry. This fix should be applied even if the machine is using OSR2, because then registry should be updated properly.

This fix has no effect for users of Novell servers and Windows 95 workstations. If problems are occurring in this network configuration, using the network requester supplied from Novell has solved the file corruption.

2. **Mixed Novell networks sharing 3.x and 4.x servers.** 4.x users accessing data on a 3.x server may experience overwriting of EOF markers. The EOF is stored on the individual workstation and not passed correctly between the two server types. Removing the mixture of server versions eliminates this problem.

3. **Faulty user hardware.** Corruption of files have been reported and tracked down to bad network cards, memory chips, and motherboards. While identifying the faulty hardware is difficult, once it has been replaced the corruption to data files is corrected

How to Restore User Resized List Box Column Widths

The resize feature on listboxes is most useful if the user specified sizing is remembered and reapplied to the list box. The following procedure uses embedded GETINI, PUTINI, and PROPERTY assignment syntax statements to save and restore the user specified formatting changes. The formatting is stored in an application specific .INI file.

1. In the Application Tree dialog, DOUBLE-CLICK on your browse procedure.

The *Procedure Properties* dialog appears.

2. Press the **Embeds** button.

The *Embedded Source* dialog appears.

3. In the **Embedded Source** dialog, DOUBLE-CLICK on the "*WindowManager Method Executable Code -- Ask -- ( )* " embed point.

The *Select Embed Type* dialog appears.

4. DOUBLE-CLICK on *SOURCE*.

The Text Editor appears, ready to accept your embedded source statements.

5. Type the following statement, then **Exit**! the Text Editor and save your changes.

```
?Browse:1{PROP:FORMAT}=|
```

```
GETINI('Preferences','?Browse:1Format',?Browse:1{PROP:FORMAT},'MyApp.INI')
```

where *?Browse:1* is the field equate label for your list box, and MyApp is the name of your .APP file. "Preferences" is the section in the .INI file where the information is stored. "?Browse:1Format" is the entry in the .INI file where the information is stored, and "?Browse:1{PROP:FORMAT}" supplies the current format string as the default in case there is no formatting information in the .INI file.

6. Set the Priority to "First". This embeds the code before the PARENT.Ask() method.

7. Highlight the "*WindowManager Method Executable Code -- Ask -- ( )* " embed point again and press **Insert.**

The **Select Embed Type** dialog appears.

8. DOUBLE-CLICK on *SOURCE*.

The Text Editor appears, ready to accept your embedded source statements.

9. Type the following statement, then **Exit**! the Text Editor and save your changes.

```
PUTINI('Preferences','?Browse:1Format',?Browse:1{PROP:FORMAT},'MyApp.INI')
```

where *?Browse:1* is the field equate label for your list box, and MyApp is the name of your .APP file.

10. Set the Priority to "Last". This embeds the code after the PARENT.Ask() method.

How to Send DDE Commands and Data to a DDE Server

Once the DDE channel is open, you can use the DDE functions to send commands, data, or requests to the server.

The example code below sends a command to Excel to open a new file and save it under a specified file name. This is a common DDE task when working with commercial applications. Often, the server application allows access to "document" functions only when you specify a document name in the DDECLIENT function. The document name must be a file that already exists.

In this particular case, to execute any "document" actions, such as entering a value in a cell, Excel (and many other applications) require the DDE channel "topic" to be the name of document. Therefore, if your application is providing new data it wants the server to save in a **new** document file, your application:

- Opens a conversation about the "System" topic.
- Sends a command asking the server to save a document file under a specified name.
- Closes the conversation.
- Opens a second conversation with the server, this time specifying the newly created file's name as the topic.
- Sends the "unsolicited" (because the server didn't ask for it) data and then tells the DDE Server (Excel) to execute commands or other requests for data that apply to the file.
- Closes the conversation.

The following therefore should execute only if the example code shown in the **How to Start a DDE Conversation** topic was successful.

```
DDEEXECUTE(Channel,'[NEW(1)]')   ! Excel's File/New command
```

The DDEEXECUTE statement takes the DDE channel number as its first parameter, and the command string as the second. Excel requires you to enclose all DDE commands in square brackets. This command creates a blank spreadsheet.

The Excel command string enclosed by the square brackets is an Excel macro statement. Excel, and many other applications allow you to send a macro statement via the DDEEXECUTE statement. In this particular case, you don't have to know the name of the open Excel file to execute the statement.

Many commercial applications with their own macro languages allow you to both record and edit macros. Use the application to make a "dry run" of the actions you need it to execute, with its macro recorder turned on. Edit the resulting macro, and use the clipboard to copy each macro statement to your embedded source in the Text Editor. Put each macro statement in the second parameter of the DDEEXECUTE statement, and you can be assured of the correct syntax for the DDE command!

2. In the next embedded source line, tell Excel to save the new (blank) sheet under a name that you specify.

```
DDEEXECUTE(Channel, '[SAVE.AS("DDE_TEST.XLS",1,"", FALSE,"",FALSE)]')
```

Knowing the name allows you to close this channel, then open another specifying the file name as the topic. Note that the Excel command string requires double-quote marks.

3. Terminate the channel started under the "System" topic.

```
DDECLOSE(Channel)          ! Close first DDE channel
```

Sending Data from Client to Server

To continue the example, to send data to Excel, you need to open another DDE conversation, this time with the newly created file name as the topic:

1. Open the DDE channel and name the file as the topic.

```
Channel = DDECLIENT('Excel','DDE_TEST.XLS')    !New channel under known filename
```

- To place data in a spreadsheet cell, use the DDEPOKE statement.

```
DDEPOKE(Channel,'R1C1','999')
```

Following the successful placement of the value in the spreadsheet, you could then send further Excel macro statements using DDEEXECUTE. This would allow you to send additional spreadsheet data, highlight a range, then tell Excel to draw a chart.

How to Set Report Group Breaks

Group breaks provide a means of grouping report data into sections and optionally displaying subheadings, subtotals, or other information associated with the subgroup. Each group consists of a set of records, all sharing the same value in the BREAK field.

In order to generate meaningful groups, the records should be sorted in the same sequence as the BREAKs are declared. In other words, when you select a sort key for your report, the key will determine the variables on which you define your group breaks.

You may also break on the common fields used to relate two files. File relationships are defined in the Data Dictionary's **Relationship Properties** dialog. Adding secondary files to your procedure gives you another logical field to break on: that is, the common field(s) linking the two files.

To create a group break**:**

1. Be sure the DETAIL band is visible; if not, press the restore button.

2. Choose **Bands** ▸ **Surrounding Break**.

3. When the cursor changes to a crosshair, CLICK in the DETAIL band.

The **Break Properties** dialog appears.

4. In the **Label** field, type a valid Clarion label to use as a name for the break.

5. In the **Variable** field, type the name of a variable to break on.

You can press the ellipsis (...) button to select a break field from the **Select Field** dialog.

6. Press the **OK** button.

This inserts the group BREAK. When the report prints, it groups together all records with the same value for the BREAK field, and prints any group HEADER and FOOTER defined for the break.

If the break variable is a global or local variable, you must be sure that the executable code updates its value, so that it can generate a group BREAK.

7. Choose **Bands** ▸ **Group Header** from the menu to define a group HEADER for the BREAK.

8. When the cursor changes to a crosshair, CLICK in the BREAK mini caption bar.

The **Page/Group Header Properties** dialog appears. Specify a field equate label and any special page breaking behavior.

9. Press the **OK** button.

This inserts the group HEADER band. You may place controls here just as in any other report band. Group footers are added similarly, using **Bands** ▸ **Group Footer** from the menu

How to Sort Reports

The sort sequence of a report is determined by a KEY or INDEX defined in the Data Dictionary's **Field/Key Definition** dialog. Keys or indexes are selected for use in this particular report procedure, using the **Table Schematic Definition** dialog. When you select a sort key for your report, the key will determine the variables on which you define your group breaks.

For example, if you select the CUS:LastNameKey as your key, then your BREAK variables should be among those fields listed as components of the CUS:LastNameKey. You can see the key's component fields in the Data Dictionary's **Field/Key Definitions** dialog.

To specify the sort sequence for your report:

1. From the **Application Tree** dialog, DOUBLE-CLICK on the report procedure name.

The **Procedure Properties** dialog appears.

2. Press the **Tables** button.

The **Table Schematic Definition** dialog appears. Use this dialog to tell the Application Generator which files and keys your report procedure will access.

3. DOUBLE-CLICK the ToDo item for your procedure.

The **Insert File** dialog appears.

4. DOUBLE-CLICK the file you wish to report from.

The **Table Schematic Definition** dialog reappears. You may specify more than one file to report on: a primary file, and secondary files. The secondary files must be related to the primary file by a common field. Adding secondary files to your procedure gives you another logical field to break on: that is, the common field(s) linking the two files.

5. Press the **Key** button, to specify which key is used for this report procedure.

The **Change Access Key** dialog appears.

6. DOUBLE-CLICK the key you want for this report.

The **Table Schematic Definition** dialog reappears.

7. Press the **OK** button.

How to Start a DDE Conversation

DDE (Dynamic Data Exchange) is a Windows Inter-Process Communication (IPC) protocol. A DDE "conversation" consists of two applications trading messages. Within the DDE conversation, one application acts as the client, the other as the server.

The application which starts the conversation, requesting data or services from the other, is the client. The contacted application is the server. The server must "register" with Windows that it has server capability.

Clarion allows you to create both DDE clients and DDE servers. An application can be both. In fact, your application can act as both a client and server at the same time, though it requires two separate DDE conversations.

Starting a DDE conversation is as easy as using the DDECLIENT function. The only "catch" is that both applications must already be running to open the channel.

The simplest way to ensure that the conversation takes place at run time is to use an IF structure. The DDECLIENT function returns zero if the server application isn't already running. Test its return value, and use the RUN statement to start the server app if it returns zero.

Many of the DDE procedures and functions require that you specify the DDE channel number, which is an integer that Windows returns when you open the DDE conversation. Create a local variable to hold the return value. Begin at the **Procedure Properties** dialog of the procedure you wish to contain the code for the DDE conversation.

To enable support for the DDE commands for your project or application, you must include the DDE.CLW file, located in the LIBSRC subdirectory.

- Create a variable to hold the DDE channel number:

1. Press the **Data** button in the **Procedure Properties** dialog.
2. Press the **Insert** button in the **Local Data** dialog.
3. Type **Channel** in the **Name** field.
4. Choose **LONG** from the **Type** dropdown list.
5. Press the **OK** button to close the **Field Properties** dialog.
6. Press the **Close** button to close the **Local Data** dialog.

Initializing the Conversation

You must embed the code to initialize the DDE conversation, starting the server application if it's not already started. Assuming a menu choice in your application begins the conversation, embed the code at a field event associated with the Accepted menu choice.

1. Choose the appropriate field event in the **Embedded Source** list.
2. Press the **Edit** button.
3. Choose the **Source** item in the **Embedded Source** dialog.
4. Press the **Add** button.
5. Type the following code, substituting the file name (without extension) of the Server application for "Excel."

```
Channel = DDECLIENT('Excel','System')! Contact Excel re System topic
IF Channel < 1                       ! If no contact made
  RUN('Excel')                       ! Attempt to start Excel
  Channel = DDECLIENT('Excel','System')    ! And try again
ELSE
  RETURN         ! Give up if no contact - add error msg!
END
```

The code example is deliberately simplistic; it would be more efficient to LOOP through the attempt to contact twice, then warn the end user of the failure.

The code attempts to open a DDE conversation with Excel named as the server. The DDECLIENT function returns a value corresponding to the channel; it doesn't matter what the channel number is. If it's less than one, it failed. You must therefore start the server, and try to open the conversation again.

The second parameter of the DDECLIENT function is the DDE "Topic." It tells the server what the DDE conversation is "about." In most cases, the topic is a file name. In this case, the code names the "System" topic, which tells Excel the conversation is not regarding a particular document file.

How to Store and Display a Graphic Image with a Memo or Blob Data Type

Memo and Blob variables are capable of storing large variable length chunks of binary data. This makes them suitable for storing graphic images. MEMOs are limited to 64K or less. BLOBS have no size limit. Storing and displaying images with Memo or Blob variables requires the following:

Storing Graphic Images in BLOBs or MEMOs

To *store* the graphic image into the MEMO or BLOB variable, channel it through an IMAGE control.

That is, assume the image resides in C:\IMAGES\IMAGE.BMP. We need to transfer the .BMP file to a CW IMAGE control, then transfer from the IMAGE control to the MEMO or BLOB variable.

1. The BLOB or MEMO variable must have the BINARY attribute.

In the Data Dictionary, use the **Field Properties** dialog's General tab to set the **Data Type** to MEMO or BLOB, then check the **Binary** box.

2. Assign the image file to an IMAGE control.

In the **Image Properties** dialog, in the **File** field, use the ellipsis (...) button to name the file containing the graphic image.

or

Assign the file name with Clarion's property syntax as follows:

```
?Image1{PROP:Text} = filename
```

3. Transfer the image from the IMAGE control to the MEMO or BLOB variable using Clarion's property syntax:

   For MEMOs: CON:TheMemo = ?Image1{PROP:ImageBits}

   For BLOBs: CON:TheBlob{PROP:Handle} = ?Image1{PROP:ImageBlob}

Displaying Graphic Images from BLOBs or MEMOs

To restore (ie display) the image from a BLOB or MEMO to an IMAGE control, you must properly define the size of the IMAGE control. The IMAGE control must either be of Default size, or of a fixed size set *after* the MEMO or BLOB data is assigned to it.

1. To set the IMAGE control to default size.

In the **Image Properties** dialog, on the **Position** tab, check the **Default** boxes for **Height** and **Width**.

2. Use Clarion's property syntax to transfer the MEMO or BLOB data to the IMAGE control.

   For MEMOs: ?Image2{PROP:ImageBits} = CON:TheMemo

   For BLOBs: ?Image2{PROP:ImageBlob} = CON:TheBlob{PROP:Handle}

3. *After* the MEMO or BLOB has been assigned to the IMAGE with property syntax, a fixed width and height may be assigned to the IMAGE Control:

```
?Image2{PROP:Width} = 92
?Image2{PROP:Height} = 88
```

How to Suppress Printing a Detail Band until Explicitly called to Print

Every Detail Band on a template generated Report procedure always prints unless you suppress printing it using a Detail Filter.

To suppress a Detail Band from printing automatically:

1. On the Band Properties dialog, name a field equate label as the detail structure's USE attribute.
2. On the Procedure Properties for the Report, press the Report Properties button, then select the Detail Filters tab
3. Highlight the band to filter out and press the Properties button.
4. Provide the expression in the Filter: field. In this case use FALSE as the expression.

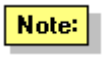
   FALSE will always be false, therefore printing the band will be suppressed.
5. Use embedded source code to explicitly print the detail band at the appropriate time:

   ```
   PRINT(RPT:MyDetailBand)
   ```

How to Synchronize your App and Dictionary

The **Synchronize** command applies the Data Dictionary's respective control specifications, including Screen Picture, Prompt, Heading, Case, Typing Mode, Flags, Justification, Initial Value, Help IDs, Messages, Tool Tips, Validity Checks, position, type of control, etc. to the controls in your application. The control attributes are applied according to the settings in the **Application Options** dialog.

With the **Application Options** dialog, you can specify whether the control's type and position change, whether blank dictionary attributes replace filled control attributes (e.g. Tool Tips or list box column headers), whether controls that are explicitly frozen, etc.



Synchronize does not change the *number* of controls on a window; therefore, synchronization does not replace an Option plus Radio buttons with a drop-down list and vice versa. In either case, the Application Generator issues a warning that no synchronization occurred for the asymmetrical controls.

Levels of Synchronization

You can synchronize:

- **A single control**: In the Window Formatter or Report Formatter, select the control, then choose **Edit** ▸ **Synchronize** from the menu (or RIGHT-CLICK and choose **Synchronize**).

If you synchronize a single control, the synchronizer ignores any #Freeze attribute for the control.

- **All the controls in a data structure**: In the Window Formatter or Report Formatter, select the data structure (WINDOW, TAB, GROUP, REPORT, DETAIL, etc.), then choose **Edit** ▸ **Synchronize** from the menu (or RIGHT-CLICK and choose **Synchronize**).
  or
  In the Window Formatter, choose **Edit** ▸ **Synchronize Window** from the menu.
  or
  In the Report Formatter, choose **Edit** ▸ **Synchronize Report** from the menu.

- **All the controls in a procedure**: In the Application Tree dialog, select a procedure, then choose **Procedure ▸ Synchronize** from the menu (or RIGHT-CLICK and choose **Synchronize**).

- **All the controls in the application**: In the Application Tree dialog, choose **Application ▸ Synchronize** from the menu.

How to Trap a Double Click on a List Box

Trapping a DOUBLE-CLICK on a list box is built into the Clarion and ABC BrowseBox templates.

To trap a DOUBLE-CLICK on a list control in hand-code:

1. Establish an ALRT(MouseLeft2) attribute on the LIST control.
2. Trap for EVENT:AlertKey on the LIST control.
3. Trap for the MouseLeft2 key code, as in the following example:

```
ACCEPT
 CASE FIELD()
 OF ?List
  CASE EVENT()
  OF EVENT:AlertKey
   IF KEYCODE() = MouseLeft2
    CurrentSel = CHOICE(?List1)
    ! Get current selection in list box
    GET(TheQueue, CurrentSel)
    ! Get corresponding data from queue
   END
  END
 END
END
```

The above code finds out what item the user DOUBLE-CLICKED on using the CHOICE() function, then uses the GET() function to retrieve the item from the QUEUE.

How to Turn Off the Tooltips in an Application.

The little tips that appear when you "hover" the cursor over a control for a few second are useful additions to a program. Some users prefer to have these options turned off. You can disable tooltips on any of three levels: Application-wide, for a single window, or for an individual control.

To turn tips off:

1. Create a global variable (for the purpose of this example, we'll call it TipFlag). Select the Global button from the Application Tree, then select the Data button. Create the variable as a BYTE. Return to the Global properties Window.

2. Choose your favorite spot and method for prompting the user whether they want the tips or not. If they select no tips, set the global variable flag = 1.

   You can set this flag through a toggle on a menu or a checkbox on a window. If desired, you can save the value to your application's INI file and read it back in during program setup (using PUTINI and GETINI).

3. To implement this for an individual control, a single window, or application-wide. Use the following steps:

   For an individual control, in the Window Formatter:

   RIGHT-CLICK on the control.

   Select the Embeds choice from the popup menu.

   In the "Preparing to process the Window" embed point, insert the following code:

```
IF TipFlag = 1
  ?mycontrol{PROP:notips} = 1
 ELSE
  ?mycontrol{PROP:notips} = 0
 END
```

   For a particular window, from the Application Tree:

   Highlight the procedure and click the Properties button. Select the

   Embeds button on the Procedure Properties dialog. Choose the

   "Preparing to process the Window" embed point and insert the following code:

```
IF TipFlag = 1
 mywindow{PROP:notips} = 1 ! where mywindow is the label of the WINDOW
ELSE
 mywindow{PROP:notips} = 0
END
```

For an entire application, from the Application Tree:

Press the **Global** button, the press the Embeds button on the **Global Properties** dialog. In the "Program Setup" embed point, insert the following code:

```
! This code must come after the any global code to get

!      the users preference for tips or no tips
       IF TipFlag = 1
         SYSTEM{PROP:notips} = 1
       ELSE
         SYSTEM{PROP:notips} = 0
       END
```

How to Use a Combo Box

This topic describes how to use a combo box without using the File Drop Combo Control template. For information on using the template, see File Drop Combo control template .

There are two ways to use a combo box--for a static list and for a list of choices from a file.

For a finite list of static choices:

1. Place a COMBO control on the window.
2. In the **Select Field** dialog, select the USE variable for it to update.
3. Press the **Cancel** button to close the **List Box Formatter** (no need to use it).
4. RIGHT-CLICK the combo box, then choose **Properties**.
5. In the **From** field, type the list of choices as a string constant (in single quotes with | separating choices), eg 'One|Two|Three'
6. Enter a numeric value into the **Drop** field.

This the number of rows displayed at one time by the drop down list.

7. Enter the correct picture token in the **Picture** field.
8. Press the **OK** button.

For a list of choices from a file:

1. On the **Procedure Properties** dailog, press the **Data** button.
2. Add a QUEUE to the local data.
3. Add a single field to the QUEUE to hold the data from the file.
4. Return to the **Procedure Properties** dialog, then press the **Window** button.
5. Place a COMBO control on the window.
6. In the **Select Field** dialog, select the USE variable for it to update.
7. Press the **Cancel** button to get out of the **List Box Formatter** (no need to use it).
8. RIGHT-CLICK then choose **Properties**.
9. Type the name of the QUEUE into the FROM field, or press the ellipsis button to select or define a QUEUE.
10. Enter a numeric value into the **Drop** field.

This the number of rows displayed at one time by the drop down list.

11. Enter the correct picture token in the **Picture** field.
12. Press the **OK** button.

13. Return to the **Procedure Properties** dialog, then press the **Embeds** button.

14. Open the Procedure Setup embed point and add code to open the file and build the COMBO control's QUEUE.

15 Open the End of Procedure embed point and add code to close the file and FREE the QUEUE.

How to Use Dropdown Lists to Lookup Records

There are two control templates which allow you to easily add the ability to lookup records from a data file and save a value to another file--the FileDrop control and the FileDropCombo.

Using the FileDrop Control Template (pick from existing list only, no updates)

This Control template scrolls through a data file and assigns the value of the selected field to the Use variable of the list control. It does not accept input. If you want the ability to enter data and select data, use the FileDropCombo Control template.

There are two different scenarios for which you can use this Control template:

- Displaying, then storing the displayed data.
- Displaying data. then storing an associated code.

Storing and Displaying the Same Data

In this scenario you want to select a value from the lookup file and store it in the Primary file. For example, a Product File with a field storing a color, with a lookup file of colors.

In this case, complete the LIST's **Action** prompts as follows:

Use The field to which the lookup value is assigned: select this field when the Application Generator prompts you with the **Select Field** dialog.

Using the List Box Formatter, populate the list with the field from the lookup file (you may populate additional fields, but be sure to populate the display field first).

Field to Fill From The field from the lookup file. This value is assigned to the Target Field.

Target Field The field to which the Fill From value is assigned. In this case this is the same as the USE variable.

Press the More Field Assignments button to automatically assign values to other fields.

Record Filter Optionally type an expression to limit the contents of the drop-down list to only those records which match the filter expression.

Default to First entry if Use Variable empty
Automatically assign the value of the first field in the list to the USE variable. The fields in the list are sorted alphabetically (unless you specify **Sort Fields**).

Displaying Text Data and Storing a Code

In this scenario you want to select a value from a textual field in the lookup file and store its associated code in the Primary file. For example, a Product File with a field storing a Location Code, with a lookup file of Locations. You want the user to select the Location from a list of descriptions, but store the Location Number in the Product file.

In this case, complete the prompts as follows:

Use Create a local variable that matches the text field. You can create this variable when the Application Generator prompts you with the **Select Field** dialog.

Using the List Box Formatter, populate the list with the text field from the lookup file (you may populate additional fields, but be sure to populate the display field first).

Field to Fill From The code field from the lookup file. This value is assigned to the Target Field.

Target Field The field to which the Fill From value is assigned.

Press the More Field Assignments button to automatically assign values to other fields.

Record Filter Optionally, type an expression to limit the contents of the drop-down list to only those records which match the filter expression.

Default to First entry if Use Variable empty
Automatically assign the value of the first field in the list to the ?USE variable. The fields in the list are sorted alphabetically (unless you specify **Sort Fields**).

List Properties:

The List Properties for this control are the same as a list; however, the **From** entry requires some explanation.

From: When placing a FileDrop Control, this field defaults to Queue:FileDrop. You should not modify this.

FileDropCombo Control Template (pick from list and add to list)

This Control template scrolls through a data file and assigns the value of the selected field to the COMBO's Use variable. It also allows adding records by typing a new value in the entry portion of the combo box.

There are two different scenarios for which you can use this Control template:

- Storing and Displaying the same data
- Displaying text data and storing a code.

Storing and Displaying the Same Data

In this scenario you want to select a value from the lookup file and store it in the Primary file. For example, a Product File with a field storing a color, with a lookup file of colors.

In this case, complete the prompts as follows:

Use The field to which the lookup value is assigned: select this field when the Application Generator prompts you with the **Select Field** dialog.

Using the List Box Formatter, populate the list with the field from the lookup file (you may populate additional fields, but be sure to populate the display field first).

Field to Fill From The field from the lookup file. This value is assigned to the Target Field.

Target Field The field to which the Fill From value is assigned. In this case this is the same as the USE variable.

Record Filter Optionally, type an expression to limit the contents of the drop-down list to only those records which match the filter expression.

Remove Duplicates Check this box to remove duplicates from the list.

Default to First entry if Use Variable empty
Automatically assign the value of the first field in the list to the USE variable. The fields in the list are sorted alphabetically (unless you specify Sort Fields).

Update Behavior

In this scenario (a lookup file with only one required field), a form is NOT needed to update the lookup file.

Allow Updates Check this box to enable updates directly from this control.

Update Procedure Leave this field blank, because no separate update (form) procedure is needed.

Displaying Text Data and Storing a Code

In this scenario, you want to select a value from a text field in the lookup file and store its associated code in the Primary file. For example, a Product File with a field storing a Location Code, with a lookup file of Locations. You want the user to select the Location from a list of descriptions, but store the Location Number in the Product file.

In this case, complete the prompts as follows:

Use Create a local variable that matches the textual field. You can create this variable when the Application Generator prompts you with the **Select Field** dialog.

Using the List Box Formatter, populate the list with the textual field from the lookup file (you may populate additional fields, but be sure to populate the display field first).

Field to Fill From The code field from the lookup file. This value is assigned to the Target Field.

Target Field The field to which the Fill From value is assigned.

Record Filter Optionally, type an expression to limit the contents of the drop-down list to only those records which match the filter expression.

Remove Duplicates Check this box to remove duplicates from the list displayed.

Default to First entry if Use Variable empty
Automatically assign the value of the first field in the list to the ?USE variable. The fields in the list are sorted alphabetically (unless you specify Sort Fields).

Update Behavior

In this scenario (a lookup file with multiple required fields), a form is needed to update the lookup file.

Allow Updates Check this box to enable updates.

Update Procedure Specify an update (form) procedure to call to add new records to the lookup file.

List Properties:

The List Properties for this control are the same as a list; however, the **From** entry requires some explanation.

From:	When placing a FileDropCombo Control, this field defaults to Queue:FileDropCombo. You should not modify this.

How to Use Pattern Pictures on a Form

The standard Windows behavior for an entry control is free-form entry. To override this behavior, you must add the MASK attribute to the WINDOW.

In the Window Formatter:

1. Make sure the Window is selected.
2. Choose **Edit ▸ Properties** (or press ENTER).

   The **Window Properties** dialog appears.
3. On the **Extra** tab, check the **Entry Patterns** box.
4. Press the **OK** button.
5. For each entry control in the window, add a Picture token to the **Picture** field of the **Entry Properties** dialog.

How to Use *Preview!*

You can quickly "preview" alternative layouts for DETAILs, HEADERs, and FOOTERs. Fonts, sizes, colors, and positions of report controls are all displayed, and you can see these effects all without actually compiling or running your report.

In the Report Formatter:

1. Choose **Preview!** to "visualize" how the printed page will appear.

The **Preview Print Details** dialog appears. This dialog lets you generate "filler" data for your report. The data have no values, but serve as placeholders, so you can get a feel for the appearance of your finished report.

2. Highlight a detail (if you have more than one) in the **Details** list then press the **Add** button.

Each press of the **Add** button populates the preview with a filler record. Add one record for a one-record-per-page type report, or add lots of records to see the effects of the page breaking, group breaking, and header and footer options you have selected.

3. Press the **OK** button.

The Report Formatter generates a preview of your report including DETAILs, HEADERs, FOOTERs, BREAKs, fonts, sizes, colors, and positions of report controls.

4. When done "previewing," choose **Band View!** to continue editing your report.

How to Use Range Limits and Filters

There are many times that you will want to view, process, or report a sub-set of records from a file. There are two ways to do this:

1. **Range Limits**
2. **Filters**

Each method has its tradeoffs. Range Limits are much faster to process, but they require that a procedure or control use a limited key as the primary key. Filters are more flexible, since they don't require any special key manipulation, but they are much slower. In any procedure that does sequential processing, you can specify a Range Limit Field, and the type of Range Limit you want to use. The types provided are:

Current Value -Value Limited Keyed Access

The key element specified in the Range Limit Field prompt is the final Fixed Key Element. With this kind of Range Limit, the value of all Fixed Key Elements are saved when the procedure is initialized. These values are used for the duration of the procedure.

Single Value-Value Limited Keyed Access

The key element specified in the Range Limit Field prompt is the final Fixed Key Element. With this kind of Range Limit, the values of all Fixed Key Elements EXCEPT the final Fixed Key Element are saved when the procedure is initialized. The final Fixed Key Element is assigned the value specified in the Range Limit value prompt. This value can be either a variable or fixed value. This value is reevaluated each time a page or entry is loaded or processed.

Range of Values -Ranged Key Access

The key element specified in the Range Limit Field prompt is the first Free Key Element. With this kind of Range Limit, the values of all Fixed Key Elements *except* the final Fixed Key Element are saved when the procedure is initialized. The Low Limit and High Limit Values are used to set the keys for sequential access and to evaluate each record read to insure it is within the valid range. These values can be either fixed values or variables. If variables are used, these variables are reevaluated each time a page or entry is loaded or processed.

There is also support in this dialog for the Higher Key Component, which allows you to specify a secondary key component for **Single Values** and **Range of Values,** and then prime the higher component values accordingly.

Browse Box Only - File Relationship - Value Limited Key Access

All Fixed Key Elements are assigned values as defined in a relationship in the data dictionary. With this kind of Range Limit, it is possible to have multiple Browse Box control templates populated on a window, and as long as the relationships are defined and used, when a parent Browse Box goes out of range, all children (and grandchildren, etc.) Browse Boxes and Controls will automatically be reconstructed.

You *must* BIND any variables or EQUATEs used in a filter expression. Add the variable to the **Hot Fields** list, and check the **Bind Field** box.

How to Use Spin Controls for Date or Time Fields

Spin Controls are commonly used for date or time entry controls. Using a SPIN control gives the end user more flexibility, allowing data entry by typing or by clicking on the up or down buttons to increment or decrement the value.

In the Window Formatter:

1. Place a SPIN control on the window by clicking on the Spin icon in the Controls toolbox and then clicking on the desired position in the window.
2. RIGHT-CLICK on the spin control and choose Properties from the popup menu.
3. In the **Use** entry box on the **General** tab, type the field's label (or press the ellipsis (...) button to select the field from the **Select Field** dialog).
4. Select the **Extra** tab, and specify the **Step** value.

   This is the amount by which the value is incremented or decremented when the Spin Control's Up or Down button is pressed. For a Date, a step value of 1 increments or decrements by one day. For a Time, a step value of 6000 increments or decrements by one minute.
5. Optionally, provide an initial value for the fields.

   You can specify an initial value in the Data Dictionary or if you are using the Form procedure template or Save button control template, you can provide an Initial value by specifying it in the **Field Priming on Insert** dialog. To specify the current date, assign the TODAY() function to a date field. To specify the current time, assign the CLOCK() function to a time field.

If you always want spin controls for these fields, specify a SPIN control as the default Window control in the Field Properties dialog in the Data Dictionary.

How to Use the Application Wizard

The following tutorial guides you through the creation of an application with the use of the Application Wizard

1. Optionally, in File Manager, choose **File** ▸ **Create Directory**, type a subdirectory name and press **OK**.

Or alternately, use the DOS prompt, and the MkDir command.

2. Choose **File** ▸ **New** (or press the  button on the toolbar)**.**

The **New** file dialog appears.

3. Choose **Application** by CLICKING on the tab.

4. Type a name for the .APP file in the **Application File** field. If you want to use the Quick Start wizard, check the box below the file list. See Using the Quick Start Wizard.

Type a legal DOS filename. Clarion automatically adds the .APP extension.

The Application Properties dialog appears. This dialog lets you define the essential files for the application.

5. Name the .DCT file the application will use in the **Dictionary File** field, or press the ellipsis (**...**) button to select the file in the **Select Dictionary** dialog.

See How to Create a Data Dictionary for information on creating your application's data dictionary. The **Select Dictionary** dialog is a standard **Open File** dialog.

The Application Generator does *not* require a data dictionary to generate an application, if you *uncheck* the **Require a dictionary** box in the **Application Options** dialog.

6. Do not rename the first procedure from MAIN.

7. Choose the **Destination Type** from the drop down list.

This defines the type of target file for your application. Choose from **Executable** (.EXE), **Library** (.LIB), or **Dynamic Link Library** (.DLL).

8. Optionally, type a name for the application's .HLP file in the **Help File** field, or use the ellipsis (**...**) button to select the file in the Open File dialog.

You can leave the field blank for now, then fill in the field at a later time.

The Application Generator lets you name the help topics in your application without determining that the help file exists. You are responsible for creating a .HLP file that contains the context strings and keywords that you optionally enter as HLP attributes for the various controls and dialogs.

9. Accept the default Clarion template in the **Application Template** field.

The selected application template controls code generation.

10. Check the **Application Wizard** box to use the wizard to create a complete application based on the selected dictionary and a few answers you specify.

11. Press the **OK** button.

The Application Wizard dialogs appear.

12. Answer the question(s) in each dialog, then press the **Next** button. On the last dialog, the **Finish** button is enabled. If you are satisfied with your answers, press the **Finish** button.

The Application Wizard creates the .APP file based on the dictionary and the answers you provided, then displays the **Application Tree** dialog for your new application.

You can control some of the wizard options in the Data Dictionary by specifying **Options** for Files, Fields, Keys, or relations. See Using Wizard Options for more information.

How to Use the Browse Procedure Wizard

From the Application Generator tree:

1. Highlight a **ToDo** Procedure in the Procedure Tree and press the **Enter** key. You can also simply press the **Insert** key, and type a procedure name in the *New Procedure* dialog.

The *Select Procedure* dialog appears.

2. Select **BrowseWizard** from the list of Procedure templates located in the *Wizards* tab.

3. Press the **Select** button. The **Procedure Wizard** dialogs appear.

4. Answer the questions in each dialog, then press the **Next** button.

5. On the last dialog, the **Finish** button is enabled. If you are satisfied with your answers, press the **Finish** button.

The Procedure Wizard creates the procedure(s) based on the dictionary file and the answers you provided, and then displays the Procedure Properties dialog for your new procedure.

You can control some of the wizard options in the Data Dictionary by specifying **Options** for Files, Fields, Keys, or relations. See Using Wizard Options for more information.

How to Use the Form Procedure Wizard

1. Highlight a **ToDo** Procedure in the Procedure Tree and press the **Enter** key. You can also simply press the **Insert** key, and type a procedure name in the *New Procedure* dialog.

The *Select Procedure* dialog appears.

2. Select **FormWizard** from the list of Procedure templates located in the *Wizards* tab, and Press the **Select** button.

The **Procedure Wizard** dialogs appear.

3. Answer the question(s) in each dialog, and then press the **Next** button. On the last dialog, the **Finish** button is enabled. If you are satisfied with your answers, press the **Finish** button.

The Procedure Wizard creates the procedure based on the dictionary file and the answers you provided, and then displays the Procedure Properties dialog for your new procedure.

You can control some of the wizard options in the Data Dictionary by specifying **Options** for Files, Fields, Keys, or relations. See Using Wizard Options for more information.

How to Use the Report Formatter - An Overview

Use the **Report Formatter** to visually design report elements--headers, totals, detail lines, graphic images, preprinted forms, etc.--on screen. The **Report Formatter** automatically generates the Clarion language source code that defines these elements.

The **Report Formatter** has six major components that help design your report: the Sample Report with Rulers and Grid Snap, the Controls Toolbox, the Fields Toolbox, the Property Toolbox, the Align Toolbox, and report Preview.

Using the Report Formatter - A Typical Procedure

The **Report Formatter** represents the four basic parts of the REPORT data structure by showing the page HEADER, DETAIL, FOOTER, and FORM as four "bands."

Here is the typical process for developing a report with the **Report Formatter**:

1. Specify the files and sort keys your report procedure will use.
2. Establish the general layout of your report: paper size, page orientation, page margins, and band positions.
3. Place constants such as report titles and logos in the header band or the form band.
4. Place variables such as page numbers and section headers in the header band.
5. Place data dictionary fields from your data files in the detail band.
6. Set a group break or breaks, with variable headers, subtotals, etc.
7. **Preview**! your report.
8. Repeat any of the above steps as necessary, to fine tune your report.
9. **Exit**! the Report Formatter.

Using the Report Formatter - Page Layout

1. Choose **View** ▸ **Page Layout View**.
2. Reposition the report bands by dragging their handles.

Bands may overlap, abut, or be separated by space.

How to Set Report Margins

The default margins for the detail print area are one inch from the left edge of the page, and two inches from the top. This setting leaves space for a HEADER at the top of the page. You specify the margins on the **Position** tab of the respective **Report Properties, Page/Group Header Properties, Detail Properties,** and **Page/Group Footer** dialogs.

- To specify the left margin, enter a value in the **X pos** box.
- To specify the top margin, enter a value in the **Y pos** box.
- To specify the height, enter a value in the **Height** box.
- To specify the width, enter a value in the **Width** box.

These values set the AT attribute for the selected report structure.

The AT attribute on report structures performs two different functions, depending upon the structure on which it is placed.

When placed on a FORM, or **page** HEADER or FOOTER the AT attribute defines the position and size on the page at which the structure is printed. The position specified by the **x** and **y** parameters is relative to the top left corner of the page.

When placed on a DETAIL, or **group** HEADER or FOOTER the print structure is printed according to the following rules (unless the ABSOLUTE attribute is also present):

- The width and height parameters of the AT attribute specify the **minimum** print size of the structure.
- The structure is actually printed at the next available position within the detail print area (specified by the REPORT´s AT attribute).
- The position specified by the x and y parameters of the structure's AT attribute is an offset from the next available print position within the detail print area.
- The first print structure on the page is printed at the top left corner of the detail print area (at the offset specified by its AT attribute).
- Next and subsequent print structures are printed relative to the ending position of the previous print structure:

If there is room to print the next structure beside the previous structure, it is printed there.

If not, it is printed below the previous.

The values contained in the AT attribute's **x, y, width,** and **height** parameters default to dialog units unless the THOUS, MM, or POINTS attribute is also present. Dialog units are defined as one-quarter the average character width by one-eighth the average character height. The size of a dialog unit is dependent upon the size of the default font for the report. This measurement is based on the font specified in the FONT attribute of the report, or the printer's default font.

Using the Report Formatter - Paper Size

1. Choose **Edit** ▸ **Report Properties**.

The **Report Properties** dialog appears.

2. Select the **Paper Size** tab.

3. Standard Sizes: choose from forty-one sizes in the **Paper Size** drop down list.

The list includes standard letter, legal, ledger, envelopes, and more.

4. Custom Sizes: choose **Other** from the **Paper Size** drop down list, then type your own width and height values.

5. Check the **Landscape** box to orient the report text parallel with the longest edges of the paper.

Using the Report Formatter - Constant Strings

1. Choose **Controls** ▸ **String**, or pick the **String** tool from the **Controls** toolbox.

2. CLICK in one of the report bands.

3. DOUBLE-CLICK on the string control you just placed.

The **String Properties** dialog appears.

4. Type the constant text**:** in the **Parameter** field, then press the **OK** button.

5. Resize the control so that it's wide enough to hold the text, by DRAGGING its right handle.

Using the Report Formatter - Variable Strings

Use variable strings to display data dictionary fields, Clarion total fields, such as Page No., Sum, Average, Count, etc., and your own calculated fields.

1. Choose **Controls** ▸ **String**, or pick the **String** tool from the **Controls** toolbox.

2. CLICK in one of the report bands (except the form band - forms cannot display variables).

3. DOUBLE-CLICK on the string control you just placed.

4. Check the **Variable String** box.

5. Type a picture token in the **Picture** field.

@n2 specifies a numeric picture for the control. @S10 specifies an alpha-numeric picture for the control.

6. For data dictionary fields or memory variables, press the **Use** field ellipsis (...) button to choose a field (or define a new field) from the **Select Field** dialog.

7. For Clarion total fields, simply choose a total type from the **Total Type** drop down list.

8. Press the **OK** button to close the **String Properties** dialog.

Using the Report Formatter - Graphic Images

1. Choose **Controls** ▸ **Image**, or pick the **Image** tool from the **Controls** toolbox.

2. CLICK in one of the report bands.

3. Resize the image control by DRAGGING its handles.

4. DOUBLE-CLICK on the image control you just placed.

The **Image Properties** dialog appears.

5. Press the **File** ellipsis (...) button to choose a graphic file from the **Select Image File** dialog.

Choose bitmaps (.BMP), metafiles (.WMF), icons (.ICO), jpeg (.JPG), etc.

How to Use the Report Procedure Wizard

1. Highlight a ToDo Procedure in the Procedure Tree and press enter.

The Select Procedure dialog appears.

2. Select **ReportWizard** from the list of Procedure templates found on the *Wizards* tab, and press the **Select** button.

The **Procedure Wizard** dialogs appear.

3. Answer the question(s) in each dialog, then press the **Next** button. On the last dialog, the **Finish** button is enabled. If you are satisfied with your answers, press the **Finish** button.

The Procedure Wizard creates the procedure based on the dictionary file and the answers you provided, and then displays the Procedure Properties dialog for your new procedure.

You can control some of the wizard options in the Data Dictionary by specifying **Options** for Files, Fields, Keys, or relations. See Using Wizard Optionsfor more information.

How to Use Windows DLLs NOT Created in Clarion

You can use Windows DLLs which have not been created with Clarion if you know the prototypes for the .DLL's procedures and functions. See the *Programmer's Guide* article on *Multi-Language Programming* for more information on this topic.

If the source language prototypes are known, then equivalent Clarion prototypes must be created and included in a CLARION program's MAP for all referenced DLL procedures and functions. Also, Clarion requires a Library (.LIB) file in the Project Tree under **Library and Object files**. This Library file entry enables the linker to resolve the procedure and function references in the .DLL.

If you have a Windows DLL (not created with Clarion) that you want to use in a CLARION program, then the following steps are required to enable the CLARION program to access the DLL's procedures and functions:

- Create Equivalent Clarion Language Prototypes.
- Create a Clarion Library (.LIB) File for the DLL.
- Reference the Library (.LIB) File in the Project System.

Create Equivalent Clarion Language Prototypes.

Prototypes for any referenced .DLL procedures and functions must be in the CLARION program's MAP structure. Procedures and functions written in a language other than Clarion can still be referenced in a Clarion program by creating an equivalent Clarion prototype. The prototypes are placed in a MODULE structure which identifies the name of the DLL's library as the MODULE parameter. For example, if the DLL name is MY.DLL then the module structure would be:

```
MODULE('MY.LIB')
```

There are several issues to consider when creating equivalent prototypes in Clarion which are dependent upon a DLL's source code language. A primary consideration is relating equivalent data types in the other language to Clarion. Equivalent data types can be determined by considering the "underlying" machine data type represented by each language data type. For example, the CLARION Language Reference identifies the Clarion data type SREAL as a "four-byte signed floating point".

The following is an example of C and C++ code data type equivalents.

```
unsigned char       ==> BYTE
short               ==> SHORT
unsigned short      ==> USHORT
long                ==> LONG
unsigned long       ==> ULONG
float               ==> SREAL
double              ==> REAL
```

A Clarion GROUP is roughly equivalent to a C or C++ *struct*. For example:

```
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 {
       unsigned long  ul1;
                           unsigned long       ul2;
} Struct1;
```

A second important prototyping consideration is the procedure/ function calling convention utilized by another language. Clarion provides support for three different calling conventions: PASCAL, C, and TopSpeed's Register Based.

Create a Clarion Library .LIB File for the DLL.

You can create a .LIB file for the DLL using the LIBMAKER.EXE utility program that comes with Clarion as one of the example programs. Simply run the program, select the .DLL and have it automatically create the .LIB file for you.

Reference the .LIB File in the Project System.

You must place the Library (.LIB) file in the Project Tree under **Library and Object files** for any Project that will use the .DLL.

How to Work with SHEET and TAB controls

The SHEET control declares a group of TAB controls that offer the user multiple "pages" of controls for the window. The multiple TAB controls in the SHEET structure define the "pages" displayed to the user.

There are several ways to use SHEETs and TABs on a WINDOW. This topic covers the two most common uses: Placing Controls within a TAB and Placing Controls "on top of" a TAB but outside of the TAB structure.

TAB structures may contain any number of controls after the TAB declaration and before its END terminator. Controls which are declared within a TAB structure are only visible when the TAB is selected. When the TAB is not selected, the controls within the TAB structure are hidden in a different manner than a HIDE statement--that is they can still be SELECTed or accept any property assignments.

Form procedures made with the Form Wizard or the Application Wizard create Browse procedures for Child files using this method--the BrowseBox control template (a LIST control) and the BrowseUpdate buttons are within the TAB structure, so they display only when the user selects the TAB.

You can place Controls "on top of" a TAB, but outside of its structure by specifying its AT() location to be in the same place as the TAB. To accomplish this in the Window Formatter, you would populate the control somewhere outside of the TAB, then drag it on top of the tab. When the formatter asks if the control should be moved into the TAB, press the **NO** button. Another method of controlling the control's placement in code is through the **Set Control Order** command on the **Edit** Menu.

A control outside of the TAB/SHEET structure is always visible. The Browse Wizard and Browse procedures made with the Application Wizard use this technique for files with more than one KEY. The LIST control is "on" the TAB, but not within the TAB structure. The Conditional Browse Box behavior is set to change the LIST's sort order based on the selected TAB. In a similar manner, the BrowseUpdate Buttons are "on" the TAB but also outside the TAB structure. This allows a single set of BUTTONs to remain active regardless of the selected TAB.

What is...?

What is a Control Template?

A control is almost anything you see on a window or a report. For example, a check box, a push button, an entry field, and a list box are all controls.

Control templates generate source code to declare controls *and* manage their associated data. For example, the BrowseBox Control template not only generates source code to declare a list box, it also generates code to load data into a QUEUE, then display the QUEUE in the list box with complete scrolling, searching, sorting, updating, and mouse-click selection capability.

Control templates can also control file I/O; for example, the SaveButton Control template can warn that changes were made if the end user tries to close the window without saving the changes to disk.

Generally, it is to your advantage to use a Control template rather than a simple control.

See Also:

Adding Control Templates

What is a Dialog Unit?

Dialog units are Windows' standard way of measuring distance on screen.

Dialog units are defined as one-quarter the average character width by one-eighth the average character height of the default font used on a window (creating a roughly square unit of measurement). The actual size of a dialog unit can vary from one window to the next, depending upon your selection of the font to use for the window. If you do not choose a specific font for a window, the Windows default font is used as the default font for the window and as the basis for the definition of the size of a dialog unit.

The purpose of using such a "floating definition" for sizing and positioning controls in Windows is to ensure that the relative design of your windows is kept despite any font change. The user can change their Windows default font in SYSTEM.INI, and Windows video drivers supplied by video board manufacturers can also change the Windows default font. Therefore, using dialog units ensures consistent relative window and control size and positioning from one computer to the next.

What is Redirection File?

Clarion's development environment sets the working directory to the one in which the current application or project file resides. Additionally, Clarion uses the redirection file (%ClarionRoot%\BIN\C6EE.RED) to keep track of directories for the various application or project components. This redirection file tells the development environment where to find files and where to create new files.

Clarion checks for a (local) C6EE.RED file in the application directory first. If it finds no local redirection file, it uses the default redirection file--installed by default to ..\BIN\C6EE.RED.

Note: Backup files are always created in the directory where the original file is located.

To edit the default redirection file, choose **Setup ▸ Edit Redirection File**. The text editor opens the ..\BIN\C6EE.RED file for editing.

Redirection File Syntax

Each line of the redirection file is in the format:

filemask = directory$_1$ [;directory$_2$]... [;directory$_n$]

The *filemask* is a file name or a file mask using the standard DOS wild card characters: * and ?.

The *directory* is a pathname identifying the directory or folder to search for the *filemask* files. The first *directory* is where any new *filemask* files are created. This is only true for files created and saved by the development environment, such as .OBJ, .DBD, .LIB, .EXE, and .CLW. The additional *directory* entries name additional search paths for existing *filemask* files.

Redirection Macros

The redirection file *directory* can contain macros. Redirection macros are labels surrounded by the percent sign (%). Whereever it encounters a redirection macro, the Clarion environment substitutes the macro's substitution value. You define redirection macros and their substitution values in the [Redirection Macros] section of the ..\BIN\C6EE.INI file. For example to define a TEMP macro add the following to C6EE.INI:

```
[Redirection Macros]
TEMP=c:TEMP
```

To use the TEMP macro, add the following to C6EE.RED:

```
*.dbd = %TEMP%\obj
```

The Clarion environment expands the redirection line to:

```
*.dbd = c:TEMP\obj
```

The ROOT Macro

The redirection file *directory* can contain the predefined %ROOT% macro. By default, the %ROOT% macro expands to the drive and path one level above that from which the environment program is executing. For example, if the environment program is in C:\C60\BIN, the environment substitutes C:\C60 for %ROOT%. The default redirection file uses the %ROOT% macro to work with Clarion's default directory structure, regardless of where you install Clarion.

You may override the %ROOT% macro's default substitution value by explicitly setting a value in the .INI file. For example:

```
[Redirection Macros]
ROOT=d:\C60
```

Redirection File Sections

The redirection file can be separated into sections that are conditionally ignored or used depending on Project System settings. The sections are of the form [NNNNN] where NNNNN is one of the following:

Section Name	Project System Switch
32	Target OS=32-bit
DEBUG	Debug Mode
RELEASE	Build Release System
DEBUG32	Debug Mode + Target OS=32-bit
RELEASE32	Build Release System + Target OS=32-bit
COMMON	none - COMMON is always used

Redirection lines within a section are only used if the section's corresponding Project System switches are true (COMMON is always true). Redirection lines without a section are always used. For example:

```
 [DEBUG32]
*.obj = c:\test32
[RELEASE]
*.obj = c:\release
[COMMON]
*.* = work
```

In this example if the project is 32-bit and the Build Release System box is checked, then .obj files are created in c:\release.

cleared, then .obj files are created in c:\test32 from line 2.

A Default Redirection File

```
[Debug32]
*.obj = %ROOT%\obj32
*.lib = %ROOT%\obj32
*.res = %ROOT%\obj32
*.rsc = %ROOT%\obj32

[Release32]
*.obj = %ROOT%\obj32\release
*.lib = %ROOT%\obj32\release
*.res = %ROOT%\obj32\release
*.rsc = %ROOT%\obj32\release

[Common]
*.dll = .;%ROOT%\bin
*.tp? = %ROOT%\template
*.trf = %ROOT%\template
*.* = .; %ROOT%\examples; %ROOT%\libsrc; %ROOT%\images; %ROOT%\template; %ROOT%\convsrc
*.lib = %ROOT%\lib
*.obj = %ROOT%\lib
*.res = %ROOT%\lib
```

Note:

The default redirection file is designed to work with Clarion's default directory structure. If you change the directory structure, you should make corresponding changes to the redirection file.

What is a Template?

Clarion templates are highly configurable, interactive, interpreted, code generation scripts. A template typically prompts you for information then generates a custom set of source code based on your responses. In addition to its prompts, many templates also add source code embed points to your application—points at which you can supply custom source code that is integrated into the template generated code. You may want to think of the template prompts as a way to define the static (compile time) characteristics of a program or procedure, and the embedded source as a way to define the changing (runtime) characteristics of a program or procedure.

Template Prompts

A template typically prompts you for information at design time. The Application Generator interprets the template and presents a dialog with all the template's prompts. You fill in the prompts, with assistance from the on-line help, to define the static (compile time) characteristics of your program or procedure. For example, fill in the Record Filter prompt to establish a filter for a BrowseBox template.

Template Embed Points

In addition to its prompts, many templates also add source code embed points to your application or procedure—points at which you can supply custom source code that is integrated into the template generated code. You can use these embed points to define the changing (runtime) characteristics of a program or procedure. For example, embed source code to hide a related listbox when there are no related records to display.

Template Benefits

Templates promote code reuse and centralized maintenance of code. They provide many of the same benefits of object oriented programming, especially reusability. In addition, templates can compliment and enhance the use of object oriented code by providing easy-to-use wrappers for complex objects. The ABC Templates and ABC Library are a prime example of this synergistic relationship between templates and objects.

Template Flexibility

You can modify templates to your specifications and store your modifications in the Template Registry. You may also add third party templates and use them in addition to, and along with, the Clarion templates. You may write your own templates too. The Template Language is documented in the *Programmer's Guide* and in the on-line help.

What is the Significance of a Double-Colon in Source Code?

The Clarion standard templates generate labels containing a double colon (::). This is simply a convention the templates use to avoid name conflicts between generated labels and your hand coded labels.

Index: