Template Language Reference

SoftVelocity

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

Trademark Acknowledgements:

Printed in the United States of America (1103)

Contents:

1 - Introduction to the Template Language

Template Language Overview

Clarion for Windows' Template Language is a flexible script language complete with control structures, user interface elements, variables, file I/O, and more. The Template Language "drives" the Application Generator both at application design time and during source code generation.

- During application design, the programmer is asked for specific information about the application being generated. These prompts for information come directly from the templates.
- During source code generation, the template is in control of the source code statements generated for each procedure in the application, and also controls what source files receive the generated code.

This process makes the Templates completely in control of the Application Generator. The benefit to the programmer of this is the complete flexibility to generate code that is directly suited to the programmer's needs.

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 Templates Are

A template is a complete set of instructions, both Template and "target" language statements, which the Application Generator uses to process the programmer's input for application customizations then generate "target" language (usually, but not limited to, Clarion language) source code.

Clarion's templates are completely reusable. They generate only the exact code required for each specific instance of its use; they do not inherit unused methods. The templates are also polymorphic, since the programmer specifies the features and functions of each template that are required for the procedure. This means one template can generate different functionality based upon the programmer's desires.

Some of the most important aspects of template functionality supported by the Template Language include:

- Support for controls (#PROMPT) that gather input from the developer, storing that input in user-defined template variables (symbols).
- Pre-defined template variables (Built-in Symbols) containing information from the data dictionary and Clarion for Windows' application development environment.
- Specialized #PROMPT entry types, which give the programmer a list of appropriate choices for such things as data file or key selection.
- Unconditional and conditional control structures (#FOR, #LOOP, #IF, #CASE) which branch source generation execution based on an expression or the contents of a symbol (variable). This allows the Application Generator to generate only the exact source code needed to produce the programmer's desired functionality in their application.
- Statements (#EMBED) that define specific points where the developer can insert (embed) or *not* insert their own source code to further customize their application.
- Support for code templates (#CODE), control templates (#CONTROL), and extension templates (#EXTENSION) that add their specific (extended) functionality to any procedure template. This makes any procedure type polymorphic, in that, the procedure can include functionality normally performed by other types of procedures.

Template code is contained in one or more ASCII files (*.TPL or *.TPW) which the Application Generator pre-compiles and incorporates into the REGISTRY.TRF file. It is this template registry file that the Application Generator uses during application design.

Once in the registry, the template code is completely reusable from application to application. It generates custom source code for each application based upon the application's data dictionary and the options selected by the programmer while working with the Application Generator.

The programmer can customize the templates in the registry (or in the *.TP* files) to fit their own specific standard design requirements. This means that each procedure template can be designed to appear exactly as the programmer requires as a starting point for their applications. Multiple "default" starting points can be created, so the programmer can have a choice of starting point designs for each procedure type.

When the programmer has customized the template source (*.TP* file), the Application Generator automatically updates the registry. When the programmer has customized the registry, the template source files can be re-generated from the registry, if necessary.

The Application Generator always makes a *copy* of the template, as stored in the registry, when creating a procedure or first populating a procedure with a code, control, or extension template. Once this copy is made, the programmer further customizes it to produce exactly the functionality required by the application for that procedure.

The template language can generate more than source code: it can even be used to create add-in utilities (see #UTILITY).

Template Types

There are four main types of templates: procedure, code, control, and extension templates.

- Procedure templates (#PROCEDURE) generate procedures in an application. This is the choice you make when asked to choose the starting point for a "ToDo" procedure in the Application Generator.
- Code templates (#CODE) generate executable code into a specific embed point. The developer can only insert them at an embed point within a procedure. A list of the available code templates appears from which to choose.
- Control templates (#CONTROL) place a related set (one or more) of controls on a procedure's window and generate the executable source code into the procedure's embed points to provide the controls' standard functionality.
- Extension templates (#EXTENSION) generate executable source code into one or more embed points to add specific functionality to a procedure that is not "tied" to any window control.

What Templates Do

The template code files contain template language statements and standard "target" language source code which the Application Generator places in your generated source code files. They also contain the prompts for the Application Generator which determine the standard customizations the developer can make to the generated code.

The programmer's response (or lack of) to the prompts "drives" the control statements that process the template language code, and produces the logic that generates the source code. The templates also contain control statements which instruct the Application Generator how to process the standard code. The function of a template is to generate the "target" language source code, customized per the programmer's response to the prompts and design of the window or report.

There are some lines of code from templates that are inserted directly into your generated source code. For example, if you accept a default Copy command menu item in your application window, the following code is inserted in your generated source exactly as it appears in the template file:

```
ITEM('&Copy'),USE(?Copy),STD(STD:Copy),MSG('Copy item to Windows clipboard')
```

Some of the standard code in the template is a mix of "target" (Clarion) language statements and template language statements. For example, when the contents of a template variable (symbol) needs to be inserted in the generated source code, the Application Generator expands the symbol to the value the application will use, as it generates the source code for the application. Within the template code, the percent sign (%) identifies a variable (symbol). In the example below, the Application Generator will fill in the field equate label for the control as it writes the source code file, substituting it for the %Control variable:

```
     SELECT(%Control)
```

To support customizing the template starting point at design time, Clarion's template language provides prompt statements that generate the template's user interface, so that the Application Generator can query the developer for the information needed to customize the application. The basic interface consists of command buttons, check boxes, radio buttons, and entry controls placed on the Procedure Properties dialog. These statements can also create custom dialog boxes to gather input from the developer. While working with the Application Generator, therefore, some of the dialogs and other interface elements the developer sees are not part of the Application Generatorrather they are produced by the template.

For example, the following statement displays a file selection dialog from the application's data dictionary, then stores the programmer's choice for a data file in a variable (symbol) called %MyFile:

```
        #PROMPT('Pick a file',FILE),%MyFile
```

It makes no difference what the programmer names the files and fields, nor what database driver is selected. The programmer picks them from a file selection dialog.

The template also contains control structures to instruct the Application Generator on how to generate the code(such as, #IF, #LOOP, #CASE). These control statements work in the same manner as Clarion language control structures.

Pre-Processing and Source Code Generation

Before allowing you to create an application using the templates, the Application Generator pre-processes the template code (.TPL and .TPW) files. The Application Generator verifies the registry is up to date by testing the time stamps and file sizes of all the template source code files.

The Application Generator utilizes the templates as stored in binary form in the registry file, as it gathers customizations from the developer with the prompts and dialogs available through the Procedure Properties dialog. The Application Generator stores the template starting point for each procedure and the customization from the programmer in the .APP file.

At source code generation time, the Application Generator processes the application's procedures as stored in the .APP file against the template, a second time. Some of the more important steps it uses to produce the source code are:

- It executes the template language control statements to process the template and the procedure's customizations in the correct order.
- It resolves the template symbols both built-in and user-defined.
- It creates the source code files and writes the source code as generated by the template, line by line, including the previously evaluated symbols.
- It evaluates embed points and writes the source code, as embedded by the developer and stored in the .APP file, in the correct location within the generated source code.

Embed Points

One of the most important template language statements is #EMBED, which defines an embed point. These extend the structure and functionality of the procedure template by allowing the programmer to add their own custom code. The embed points indicate "targets" at which the developer can add their own custom code to the generated source. These are also the "targets" for the source code generated by control and extension templates.

Each procedure template allows for a certain number of default points at which embeds are allowed. These are typically points which coincide with messages (events) from the operating environment (Windows), such as when the end user moves focus from or to a field. The template programmer can add to, or subtract from, the list.

When the developer customizes the template, pressing the Embeds button in the Procedure Properties dialog provides access to all the embed points available in a procedure. The Actions popup menu selection in the Window Formatter also provides access to the embed points for a specific control.

The developer adds custom code; either hand coded from scratch in the editor, or created with a code template at the embed point. The embed points are also the points into which control templates and extension templates generate executable code to support their functionality.

The Application Generator stores the embed point's code (no matter what its origination) in the .APP file. At code generation time, the Application Generator processes the template, producing source; when it reaches an embed point, it places the developer's code, line by line, into the generated source code document.

Template Prompts

Input Validation Statements and Prompt Entry types place controls on the Procedure Properties window or Actions dialog, which the developer sees when using the template to design an application. These range from a simple string telling the Developer what to do (#DISPLAY), to command buttons, check boxes, or radio buttons. There are also specialized entry types that provide the programmer a list of choices for input, such as the data fields in the dictionary.

Standard Windows controls can be used to get information fromn the programmer on the Procedure Properties window, the Actions dialog, or custom prompt dialogs. The common control typesentry field, check box, radio button, and drop-down listare all directly supported via the #PROMPT statement.

#PROMPT places the prompt, the input control, and the symbol in a single statement. The general format is the #PROMPT keyword, the string to display the actual prompt, a variable type for the symbol, then the symbol or variable name. The Application Generator places the prompt and the control in the Procedure Properties or Actions dialog (depending on whether the prompt comes from a the procedure template or a code, control, or extension template). When the developer fills the control with a value, then closes the dialog, the symbol holds the value.

The #BUTTON statement provides additional "space" for developer input when there is more developer input required than can fit in the one dialog. This places a button in the dialog, which displays an additional custom dialog when pressed. The additional dialogs are called "prompt pages."

#ENABLE allows prompts to be conditionally enabled based on the programmer's response to some other prompt. #BOXED supports logical grouping of related prompts. Once the programmer has input data into a prompt, the #VALIDATE statement allows the template to check its validity.

These tools provide a wide range of flexibility in the type of information a template can ask the programmer to provide. They also provide multiple ways to expedite the programmer's job, by providing "pick-lists" from which the programmer may choose wherever appropriate.

Data Dictionary Interface

The templates use information from the Data Dictionary extensively to generate code specifically for the declared database. There are several symbols that specifically give the templates access to all the declarations: %File, %Field, %Key, and %Relation. These, and all the symbols related to them, give the templates access to all the ifnormation in the Data Dictionary.

Pay special attention to the %FileUserOptions, %FieldUserOptions, %KeyUserOptions, and %RelationUserOptions symbols. These are the symbols that contain the values the user enters in the **User Options** text control on the **Options** tab of the **File Properties**, **Field Properties**, **Key Properties**, and **Relation Properties** dialogs. This can be a powerful tool to customize any output from the Data Dictionary.

The best way to use these %UserOptions symbols is to set them up so the user enters their custom preferences which your template supports in the for of attributes with parameters, with each attribute separated by a comma. This gives them the same appearance as Clarion language data structure attributes. By doing this, you can use the EXTRACT built-in template procedure to get the value from the user. For example, if the user enters the following in a **User Options** for a field:

```
MYCUSTOMOPTION(On)
```

The template code can parse this using EXTRACT:

```
#IF(EXTRACT(%FieldUserOptions,'MYCUSTOMOPTION',1) = On)
  #!Do Something related to this option being turned on
#ENDIF
```

This is a very powerful tool, which allows for infinite flexibility in the way your custom templates generate source code.

Template Structure

Template Source Format

The structure of the ASCII template source file is different than the structure of a Clarion source file. To read the ASCII source for a template, start out with the following guidelines:

- Any statement beginning with a pound symbol (#) identifies a template language statement.
- A percent sign (%) before an item within any statement (template or "target" language) identifies a template symbol (variable), which the Application Generator processes at code generation time.
- Any statement that begins without the pound (#) or percent (%) is a "target" language statement which is written directly into a source code file.

The template files are organized by code sections that terminate with the beginning of the next section or the end of the file. The template code generally divides into ten sections.

- #TEMPLATE begins a template set (template class). This is the first statement in the template set (required) which identifies the template set for the registry.
- #APPLICATION begins the source generation control section. This is the section of the template that controls the "target" language code output to source files, ready to compile. One registered template set must have a #APPLICATION section.
- #PROGRAM begins the global section of the generated source code, the main program module. One registered template set must have a #PROGRAM section.
- #MODULE begins a template section that generates the beginning code for a source code module other than the global (program) file. One registered template set must have a #MODULE section.
- #PROCEDURE begins a procedure template. This is the fundamental "target" language procedure generation template.
- #GROUP begins a reusable statement group containing code which may be #INSERTed into any other section of the template. This is the equivalent of a template language procedure.
- #CODE begins a code template section which generates executable code into a specific embed point. The developer can only insert them at an embed point within a procedure. A list of the available code templates appears from which to choose.
- #CONTROL begins a control template. Control templates place a related set (one or more) of controls on a procedure's window and generate the executable source code into embed points that provides the controls' standard functionality.

- #EXTENSION begins an extension template. Extension templates generate executable source code into one or more embed points of a procedure to add specific functionality to the procedure that is not "tied" to any window control.
- #UTILITY begins a utility execution section. This is an optional section of the template that performs a utility function, such as cross-reference or documentation generation. This is similar to #APPLICATION in that it generates output to ASCII files.

A template set must have a #TEMPLATE section to name the set for registration in the REGISTRY.TRF template registry file. At least one registered template set must have #APPLICATION, #PROGRAM, and #MODULE sections.

The Template Registry File

The Template Registry file (REGISTRY.TRF) is a specialized data repository that stores template code and defaults in binary form. All the template elements available in the Application Generator come from the registry. As you add elements from the template into your application, the Application Generator retrieves the code from the registry then stores it along with your customizations, in the .APP file.

Storing the templates in a binary registry provides these advantages:

- Quick design-time performance.
- The ability to update the defaults in the registry using standard application development tools (such as the Window Formatter). For example, you can modify a procedure template's default window without writing template source code.

The sources for the REGISTRY.TRF are the template code files (.TPL and .TPW) which are installed in the TEMPLATE subdirectory. The Application Generator can read and register .TPL files, adding it to the template registry tree. The .TPW files usually contain additional procedure or code template source, which is processed along with the .TPL file by the #INCLUDE statement in the .TPL file. This allows the template author to logically separate disparate template components.

The default template file for Clarion for Windows is CW.TPL. This file uses the #INCLUDE statement to specify processing the other .TPW files which appear in the \CW\TEMPLATE directory.

Customizing Default Templates

There are two methods for customizing the templates:

- You can edit the template source code in the .TPL and .TPW files.

It is always a good idea to make a backup copy before making any modifications to the shipping templates.

When directly editing the template source code, you can change the type of source code it generates, or the logic it uses to generate the code. This is how you can make your templates generate source code the way you would write it if you were hand-coding the application.

You can also extend the functionality of the templates by adding your own features. For example, you may want to add prompts to each procedure template that allow you to generate a "comment block" at the beginning of each procedure containing procedure maintenance comments from the programmer maintaining the application.

Adding the following code to the end of any existing template set acccomplishes this modification:

```
#EXTENSION(CommentBlock,'Add a comment block to the procedure'),PROCEDURE
  #PROMPT('Comment Line',@S70),%MyComment,MULTI('Programmer Comments')
#ATSTART
  #FOR(%MyComment)
  !%MyComment
  #ENDFOR
#ENDAT
```

This code adds an extension template that is available for any procedure in the application. When you design your procedure, add the CommentBlock extension template to the procedure, then add comments to the Comment Line prompt each time you modify the procedure. At source generation time, each comment line will appear following an exclamation point (!). The block of comments appears in the code just before the PROCEDURE statement.

If you want this extension to be used in all the procedures you write, go into the Template Registry and add the extension to all the default procedures for each procedure template. This way, you can make sure it is always used, and you can even place its prompts on the Procedure Properties dialog by checking the Show on Properties box as you add the extension to the procedure template.

Once you make the changes, either choose the **Setup > Template Registry** menu selection, open an existing application, or create a new application. Make sure the Re-register When Changed box is checked in the Registry Options dialog. The Application Generator automatically pre-processes the templates to update the registry when you have made changes to the template code files.

- You can add to or edit the default user interface procedure template elements such as the standard window designs and report layouts, or your standard global and local data variables using the Template Registry.

When you highlight a procedure template in the Template Registry and press the Properties button, the Procedure Properties dialog appears, without all the custom prompts you would normally see when developing an application. Any button which is not dimmed in the Template Registry is available to you to create the default starting point for the procedure.

You can set up the procedure for the starting point that will get you furthest toward a complete procedure while requiring the least amount of customization from you at application design time. If the procedure allows it, you may use the window and report formatters, or define additional data, by pressing the appropriate buttons.

Once you've customized your template registry, you can also export your customizations to template source code files. This is useful for sharing your customizations with other developers.

To update the template source code with the customizations made in the Template Registry, press the Regenerate button in the Template Properties dialog. This updates the .TPL and .TPW files with the changes made.

Adding New Template Sets

Adding another set of templates, whether from a third-party vendor or templates you have written yourself, is a very simple process. There is only one requirement for the new template set; a #TEMPLATE statement to identify the set for the template registry. Of course, it also needs to have the specific procedure, code, control, and extension templates to add to the template registry.

For example, the following code is completely valid as a template set with nothing else added:

```
#TEMPLATE(PersonalAddOns,'My personal Template set')
#CODE(ChangeProperty,'Change control property')
  #PROMPT('Control to change',CONTROL),%MyField,REQ
  #PROMPT('Property to change',@S20),%MyProperty,REQ
  #PROMPT('New Value',@S20),%MyValue,REQ
%MyField{%MyProperty} = '%'MyValue' #<!Change the %MyProperty of %MyField
```

When you register this template set, it will appear in the template registry as Class PersonalAddOns containing just the ChangeProperty code template.

Once a template set is registered in the template registry, all its components are completely available to the programmer for their application development, along with all the components of all other registered template sets. This allows the programmer the flexibility to "mix-and-match" their components during development.

For example, the programmer could create a procedure from a procedure template in the standard Clarion template set, populate it with a control template from a third-party vendor, insert a code template into an embed point from another third-party vendor, then add an extension template from their own personally written template set. At source generation time, all these separate components come together to create a fully functional procedure that performs all the tasks required by the programmer (and nothing else). This is the real power behind Clarion's Template-oriented programming!

2 - Template Organization

This area summarizes the template statements by function.

Template Code Sections

#TEMPLATE (begin template set)
#SYSTEM (template registration and load)
#APPLICATION (source generation control section)
#PROGRAM (global area)
#MODULE (module area)
#PROCEDURE (begin a procedure template)
#GROUP (reusable statement group)
#UTILITY (utility execution section)
#CODE (define a code template)
#CONTROL (define a control template)
#EXTENSION (define an extension template)

Embed Points

#EMBED (define embedded source point)
#AT (insert code in an embed point)
#PRIORITY (set new embed priority level)
#ATSTART (template initialization code)
#ATEND (template reset code)
#CONTEXT (set template code generation context)
#EMPTYEMBED (generate empty embed point comments)
#POSTEMBED (generate ending embed point comments)
#PREEMBED (generate beginning embed point comments)

Template Code Section Constraints

#WHERE (define #CODE embed point availability)
#RESTRICT (define section use constraints)
#ACCEPT (section valid for use)
#REJECT (section invalid for use)

Default Data and Code

#WINDOWS (default window structures)
#REPORTS (default report structures)
#LOCALDATA (default local data declarations)
#GLOBALDATA (default global data declarations)
#DEFAULT (default procedure starting point)

Symbol Management Statements

#DECLARE (declare a user-defined symbol)
#ALIAS (access a symbol from another instance)
#EQUATE (declare and assign value to a user-defined symbol)
#ADD (add to multi-valued symbol)
#DELETE (delete a multi-valued symbol instance)
#DELETEALL (delete multiple multi-valued symbol instances)
#PURGE (delete all single or multi-valued symbol instances)
#CLEAR (clear single-valued symbol)
#FREE (free a multi-valued symbol)
#FIX (fix a multi-value symbol)
#FIND (super-fix multi-value symbols)
#SELECT (fix a multi-value symbol)
#POP (delete and re-fix a multi-value symbol)
#SET (assign value to a user-defined symbol)
#UNFIX (unfix a multi-value symbol)

Programmer Input and Validation Statements

#PROMPT (prompt for programmer input)
#VALIDATE (validate prompt input)
#ENABLE (enable/disable prompts)
#BUTTON (call another page of prompts)
#WITH (associate prompts with a symbol instance)
#FIELD (control prompts)
#PREPARE (setup prompt symbols)

Display and Formatting Statements

#BOXED (prompt group box)
#DISPLAY (display-only prompt)
#IMAGE (display graphic)
#SHEET (declare a group of #TAB controls)
#TAB (declare a page of a #SHEET control)

Source Generation Statements

#GENERATE (generate source code section)
#ABORT (abort source generation)
#SUSPEND (begin conditional source)
#RELEASE (commit conditional source generation)
#RESUME (delimit conditional source)
#? (conditional source line)
#QUERY (conditionally generate source)

External Code Execution Statements

#RUN (execute program)
#SERVICE (internal use only)
#RUNDLL (execute DLL procedure)

Template Logic Control Statements

#FOR (generate code multiple times)
#IF (conditionally generate code)
#LOOP (iteratively generate code)
#BREAK (break out of a loop)
#CYCLE (cycle to top of loop)
#CASE (conditional execution structure)
#INDENT (change indentation level)
#INSERT (insert code from a #GROUP)
#CALL (insert code from a #GROUP, without indention)
#INVOKE (insert code from a named #GROUP)
#RETURN (return from #GROUP)

File Management Statements

#CREATE (create source file)
#OPEN (open source file)
#CLOSE (close source file)
#READ (read one line of a source file)
#REDIRECT (change source file)
#APPEND (add to source file)
#SECTION (define code section)
#REMOVE (delete a source file)
#REPLACE (conditionally replace source file)
#PRINT (print a source file)

Miscellaneous Statements

#! (template code comments)
#$ (embededitor comment)
#< (aligned target language comments)
#ASSERT (evaluate assumption)
#CLASS (define a formula class)
#COMMENT (specify comment column)
#DEBUG (toggle debug generation)
#DEFINE (user defined macro)
#ERROR (display source generation error)
#EXPAND (expand a user defined macro)
#EXPORT (export symbol to text)
#HELP (specify template help file)
#IMPORT(import from text script)
#INCLUDE (include a template file)
#MESSAGE (display source generation message)
#PDEFINE (add #pragma to project)
#PROTOTYPE (procedure prototype)
#PROJECT (add file to project)
#TYPEMAP (map EQUATE to date type)

Built-in Template Procedures

CALL (call a #GROUP as a function)
EXTRACT (return attribute)
EXISTS (return embed point existence)
FILEEXISTS (return file existence)
FULLNAME (return file path)
INLIST (return item exists in list)
INSTANCE (return current instance number)
INVOKE (call a named #GROUP as a function)
ITEMS (return multi-valued symbol instances)
LINKNAME (return mangled procedure name)
QUOTE (replace string special characters)
REGISTERED (return template registration)
REPLACE (replace attribute)
SEPARATOR (return attribute string delimiter position)
SLICE (return substring from string)
UNQUOTE (remove string special characters)
VAREXISTS (return symbol existence)

Built-in Template Procedures

CALL (call a #GROUP as a function)

CALL(*symbol* [, *parameters*] **)**

CALL	Calls a #GROUP as a function.
symbol	A symbol that names a #GROUP section in the current template set.
parameters	The parameters passed to the #GROUP. Each parameter must be separated by a comma. All parameters defined for the #GROUP must be passed; they may not be omitted.

The **CALL** procedure places the return value from the #GROUP named by the *symbol* into the expression containing the CALL.

The *parameters* passed to the #GROUP fall into two categories: value-parameters and variable-parameters. Value-parameters are declared by the #GROUP as a user-defined symbol, while variable-parameters are declared by the #GROUP as a user-defined symbol with a prepended asterisk (*). Either a symbol or an expression may be passed as a value-parameter. Only a symbol may be passed as a variable-parameter.

Return Data Type: STRING

Example:

```
#SET(%SomeGroup,CALL(%SelectAGroup))  #!Call %SelectAGroup to get a #GROUP name
#INVOKE(%SomeGroup)                   #!Insert either %TrueGroup or %FalseGroup

#GROUP(%SelectAGroup)
#IF (%SomeCondition)
 #RETURN('%TrueGroup(Clarion)')
#ELSE
 #RETURN('%FalseGroup(Clarion)')
#ENDIF
```

See Also:

#GROUP
INVOKE
#RETURN
#CALL
#INVOKE

EXTRACT (return attribute)

EXTRACT(*string, attribute* [, *parameter*])

EXTRACT	Returns the complete form of the specified *attribute* from the property *string* symbol.
string	The symbol containing the properties to parse.
attribute	A string constant or symbol containing the name of the property to return.
parameter	An integer constant or symbol containing the number of the property's parameter to return. Zero (0) returns the entire parameter list (without the *attribute*). If omitted, the *attribute* and all its *parameters* are returned.

The **EXTRACT** procedure returns either the complete form of the specified *attribute* from the attribute *string* symbol, or just the specified *parameter*. This is useful if no built-in symbol exists for the particular attribute you need.

Return Data Type: STRING

Example:

```
#SET(%MySymbol,EXTRACT(%ControlStatement,'DROPID')   #!Return DROPID attribute
#SET(%MySymbol,EXTRACT(%ControlStatement,'DROPID',0) #!Return all DROPID parameters
```

See Also: REPLACE

EXISTS (return embed point existence)

EXISTS(*symbol*)

EXISTS	Returns TRUE if the embedded source code point is available for use.
symbol	The *identifier* symbol for a #EMBED embedded source code point.

The **EXISTS** procedure returns true ('1') if the embedded source code point is available for use, at design-time only. If the embedded source code point is not available for use, EXISTS returns false (''). An embedded source code point is available for use if the section containing it is being used. This means that all #EMBEDs in the #PROCEDURE section, and all #GROUP sections referenced in the #PROCEDURE, are always available. #EMBEDs in a #CONTROL, #CODE, or #EXTENSION section are available only if the section is being used.

Return Data Type: LONG

Example:

```
#IF(EXISTS(%CodeTemplateEmbed) = %True)
  !Generate some source
#ENDIF
```

FILEEXISTS (return file existence)

FILEEXISTS(*file*)

FILEEXISTS	Returns TRUE if the *file* is available on disk.
file	An expression containing the DOS filename.

The **FILEEXISTS** procedure returns true ('1') if the *file* is available on disk. If the *file* is not available, FILEEXISTS returns false ('').

Return Data Type: LONG

Example:

```
#IF(FILEEXISTS(%SomeFile))
  #OPEN(%SomeFile)
  #READ(%SomeFile)
  !some source
#ENDIF
```

FULLNAME (return file path)

FULLNAME(*file*, [*option*])

FULLNAME	Returns the fully qualified path of the specified *file* on disk.
file	A string constant or expression containing the filename (including extension).
option	An integer constant, variable, or expression that evaluates to a 2. Allows FULLNAME to return the path where the file will be created.

The **FULLNAME** procedure returns the fully qualified path of the specified *file* on disk. If the optional parameter is used, FULLNAME will return the path where the file will be created.

Return Data Type: STRING

Example:

```
#RUNDLL('Process File','MyModule.DLL',FULLNAME(%ChosenFile))
#EQUATE(%WCFName, FULLNAME('WIZARD.WCF', 2))
```

INLIST (return item exists in list)

INLIST(*item, symbol*)

INLIST	Returns the instance number of the *item* in the *symbol*.
item	A string constant or symbol containing the name of the item to return.
symbol	A multi-valued symbol that may contain the item.

The **INLIST** procedure returns the instance number of the *item* in the *symbol*. If the *item* is not contained in the *symbol*, INLIST returns zero (0).

Return Data Type: LONG

Example:

```
#IF(INLIST('?MyControl',%Control))
  !Generate some source
#ENDIF
```

INSTANCE(return current instance number)

INSTANCE(*symbol*)

INSTANCE	Returns the current instance number to which the *symbol* is fixed.
symbol	A multi-valued symbol.

The **INSTANCE** procedure returns the current instance number to which the *symbol* is fixed. If no #FIX or #FOR has been issued for the *symbol*, INSTANCE returns zero (0).

Return Data Type: LONG

Example:

```
#DELETE(%Control,INSTANCE(%Control))     #!Delete current instance
```

INVOKE (call a named #GROUP as a function)

INVOKE(*symbol* [, *parameters*] **)**

INVOKE	Calls a named #GROUP as a function.
symbol	A symbol that contains the name of a #GROUP section in the current template set.
parameters	The parameters passed to the #GROUP. Each parameter must be separated by a comma. All parameters defined for the #GROUP must be passed; they may not be omitted.

The **INVOKE** procedure places the return value from the #GROUP named in the *symbol* into the expression containing the INVOKE.

The *parameters* passed to the #GROUP fall into two categories: value-parameters and variable-parameters. Value-parameters are declared by the #GROUP as a user-defined symbol, while variable-parameters are declared by the #GROUP as a user-defined symbol with a prepended asterisk (*). Either a symbol or an expression may be passed as a value-parameter. Only a symbol may be passed as a variable-parameter.

Return Data Type: STRING

Example:

```
#SET(%SomeGroup,CALL(%SelectAGroup))    #!Call %SelectAGroup to get a #GROUP name
#SET(%AnotherGroup,INVOKE(%SomeGroup))  #!Call Selected Group for another #GROUP name
#INVOKE(%AnotherGroup)                  #!Insert either %DoTrueGroup or %DoFalseGroup

#GROUP(%SelectAGroup)
#IF (%SomeCondition)
  #RETURN('%TrueGroup(Clarion)')
#ELSE
  #RETURN('%FalseGroup(Clarion)')
#ENDIF

#GROUP(%TrueGroup)
  #RETURN('%DoTrueGroup(Clarion)')

#GROUP(%FalseGroup)
  #RETURN('%DoFalseGroup(Clarion)')
```

See Also: #GROUP, CALL, #RETURN, #CALL, #INVOKE

ITEMS (return multi-valued symbol instances)

ITEMS(*symbol*)

ITEMS	Returns the number of instances contained by the *symbol*.
symbol	A multi-valued symbol.

The **ITEMS** procedure returns the number of instances contained by the *symbol*.

Return Data Type: LONG

Example:

```
#DELETE(%Control,ITEMS(%Control))  #!Delete last instance
```

LINKNAME (return mangled procedure name)

LINKNAME(*prototype*, **[[***classname***], [***interfacename***]]**)

LINKNAME	Returns the mangled procedure name for the linker.
prototype	A string constant or symbol containing the prorotype of the procedure as it appears in the MAP.
class name	A string constant or symbol containing the name of the class the procedure is using.
interface name	A string constant or symbol containing the name of the interface that the class has implemented.

The **LINKNAME** procedure returns the mangled procedure name for the linker. If a *class name* or *interface name* is specified it should not be added as a parameter to the prototype.

Return Data Type: STRING

Example:

```
  %(LINKNAME(%Procedure & %Prototype)) @%ExpLineNumber
```

QUOTE (template - replace string special characters)

QUOTE(*symbol*)

QUOTE	Expands the *symbol's* string data, "doubling up" single quotes ('), and all un-paired left angle brackets (<) and left curly braces ({) to prevent compiler errors.
symbol	The symbol containing the properties to parse.

The **QUOTE** procedure returns the string contained in the symbol with all single quotes ('), un-paired left angle brackets (<), and un-paired left curly braces ({) "doubled up" to prevent compiler errors. This allows the user to enter string constants containing apostrophes, and filter expressions containing less than signs (<) without requiring that they enter two of each.

Return Data Type: STRING

Example:

```
#PROMPT('Filter Expression',@S255),%FilterExpression
#SET(%ValueConstruct,QUOTE(%FilterExpression))
#!Expand single quotes and angle brackets
```

See Also: UNQUOTE

REGISTERED (return template registration)

REGISTERED(*template*)

REGISTERED	Returns TRUE if the *template* is available for use.
template	A string constant containing the name of the #PROCEDURE, #CONTROL, #EXTENSION, #CODE, or #GROUP whose availability is in question.

The **REGISTERED** procedure returns true ('1') if the *template* is available for use, at design-time only. If the *template* is not available for use, REGISTERED returns false ('').

Return Data Type: LONG

Example:

```
#IF(NOT REGISTERED('BrowseBox(Clarion)'))
  #ERROR('BrowseBox not registered')
#ENDIF
```

REPLACE (replace attribute)

REPLACE(*string, attribute, new value* [, *parameter*])

REPLACE	Finds the complete form of the specified *attribute* from the property *string* symbol and replaces it with the *new value*.
string	The symbol containing the properties to parse.
attribute	A string constant or symbol containing the name of the property to find.
new value	A string constant or symbol containing the replacement value for the *attribute*.
parameter	An integer constant or symbol containing the number of the property's parameter to affect. Zero (0) affects the entire parameter list (without the *attribute*). If omitted, the *attribute* and all its *parameters* are affected.

The **REPLACE** procedure replaces either the complete form of the specified *attribute* from the attribute *string* symbol, or just the specified *parameter* with the *new value*. It returns the modified *string*.

Return Data Type: STRING

Example:

```
#SET(%ValueConstruct,REPLACE(%ValueConstruct,'MSG',''))  #!Remove MSG attribute
```

See Also: EXTRACT

SEPARATOR (return attribute string delimiter position)

SEPARATOR(*string, start*)

SEPARATOR	Returns the position of the next comma in the attribute *string*.
string	A string constant or symbol containing a comma delimited list of attributes.
start	An integer constant or symbol containing the starting position from which to seek the next comma.

The **SEPARATOR** procedure returns the position of the next comma in the attribute *string* from the *start* position. This procedure correctly processes nested quotes within the *string* so that commas in string constants do not cause it to return an incorrect position.

Return Data Type: LONG

Example:

```
#SET(%MySymbol,SEPARATOR(%ControlStatement,1))  #!Return first comma position
```

SLICE (return substring from string)

SLICE(*expression, start, end*)

SLICE	Returns a substring portion of the *expression* string.
expression	A string constant, symbol, or expression containing the string from which to return a substring portion.
start	An integer constant or symbol containing the starting position (inclusive) within the *expression* from which to extract the substring.
end	An integer constant or symbol containing the ending position (inclusive) within the *expression* from which to extract the substring.

The **SLICE** procedure returns the substring portion of the *expression* string identified by the *start* and *end* position values. This is equivalent to the Clarion language string slicing operation.

Return Data Type: STRING

Example:

```
#SET(%MySymbol,SLICE('ABCDE,2,4))       #!Return 'BCD'
```

UNQUOTE (remove string special characters)

UNQUOTE(*symbol*)

UNQUOTE	Contracts the *symbol's* string data, "singling up" doubled single quotes (''), and all doubled un-paired left angle brackets (<<) and left curly braces ({{).
symbol	The symbol containing the properties to parse.

The **UNQUOTE** procedure returns the string contained in the symbol with all doubled single quotes (''), doubled un-paired left angle brackets (<<), and un-paired left curly braces ({{) "singled up" (returned to single instances instead of double instances of each character). This is the direct opposite to the QUOTE procedure.

Return Data Type: STRING

Example:

```
#PROMPT('Filter Expression',@S255),%FilterExpression
#SET(%ValueConstruct,QUOTE(%FilterExpression)
#!Expand single quotes and angle brackets

#SET(%SingledValue,UNQUOTE(%ValueConstruct))
#!Contract single quotes and angle brackets
```

See Also: QUOTE

VAREXISTS (return symbol existence)

VAREXISTS(*symbol*)

EXISTS	Returns TRUE if the *symbol* is available for use.
symbol	The symbol whose existence is in question.

The **VAREXISTS** procedure returns true ('1') if the *symbol* is available for use, at design-time only. If the *symbol* is not available for use, VAREXISTS returns false ('').

Return Data Type: LONG

Example:

```
#IF(NOT VAREXISTS(%MySymbol))
  #DECLARE(%MySymbol)
#ENDIF
```

See Also: #DECLARE

3 - Complete Template Statement Alpha Listing

#! (template code comments)

#! *comments*

#!	Initiates Template Language comments.
comments	Any text.

#! initiates Template Language comments. All text to the right of the #! to the end of the text line is ignored by the Template file processor. #! comments are not included in the generated source code.

Example:

```
#! These are Template comments which
#! will not end up in the generated code
```

#$ (embededitor comment)

#$ *embeditor text*

#$	Initiates a comment in the embeditor.
embededitor text	Any comment text to display in the embeditor.

#$ initiates a text string which is displayed in the embeditor window only. This can be used to give the user more information about the generated procedure code and explain possibilities for embedded code within embed points.

Example:

```
#$ More detailed information on embed point
#$ Field priming code (usually for Insert) goes here
```

#? (conditional source line)

#?*statement*

#?	Defines a single line of source code generated only if #RELEASE commits the conditional source section.
statement	A single line of target language code. This may contain template symbols.

The **#?** statement defines a single line of source code that is generated only if a #RELEASE statement is encountered. This allows empty unnecessary "boiler-plate" code to be easily removed from the generated source.

A #EMBED that contains source to generate performs an implied #RELEASE. Any generated source output also performs an implied #RELEASE. Therefore, an explicit #RELEASE statement is not always necessary. The #? statement defines an individual conditional source line that does not perform the implied #RELEASE. When a #RESUME is executed without the output to the file being released, any conditional lines of code are un-done back to the matching #SUSPEND.

Example:

```
ACCEPT                        #!Unconditional source line
#SUSPEND
  #?CASE SELECTED()           #!Conditional source line
  #FOR(%ScreenField)
    #SUSPEND
  #?OF %ScreenField           #!Conditional source line
    #EMBED(%ScreenSetups,'Control selected code'),%ScreenField
    #RESUME
  #?END                       #!Conditional source line
#RESUME
#SUSPEND
  #?CASE EVENT()              #!Conditional source line
  #SUSPEND
  #?OF EVENT:AlertKey         #!Conditional source line
    #SUSPEND
    #?CASE KEYCODE()          #!Conditional source line
      #FOR %HotKey
        #RELEASE
    #?OF %HotKey              #!Conditional source line
      #EMBED(%HotKeyProc,'Hot Key code'),%HotKey
      #ENDFOR
    #?END                     #!Conditional source line
    #RESUME
  #?END                       #!Conditional source line
#RESUME
END                           #!Unconditional source line
```

See Also: #SUSPEND, #RELEASE, #RESUME

#< (aligned target language comments)

<_#comments_

#<	Initiates an aligned target language comment.
comments	Any text. This must start with the target language comment initiator.

#< initiates a target language comment which is included in the generated source code. The comment is generated at the column position specified by the #COMMENT statement. If the column position is occupied, the comment is appended one space to the right of the generated source code statement. Any standard target language syntax comments without a preceding #< are included in the generated code at whatever column position they occupy in the template.

Example:

```
#COMMENT(50)
#!   This Template file comment will not be in the generated code
#<!  This is a Clarion comment which appears in the generated code in column 50
 !   This Clarion comment appears in the generated code in column 2
#<// This is a C++ comment which appears in the generated code beginning in column 50
 //  This C++ comment appears in the generated code in column 2
```

See Also: #COMMENT

#ABORT (abort source generation)

#ABORT

The **#ABORT** statement immediately terminates source generation by the previous #GENERATE statement. #ABORT may be placed in any template section.

Example:

```
#IF(%ValidRangeKey=%Null)
  #ERROR(%Procedure & ' Range Error: The range field is not in the primary key!')
  #ABORT
#ENDIF
```

See Also: #GENERATE

#ACCEPT (section valid for use)

#ACCEPT

The **#ACCEPT** statement terminates #RESTRICT processing, indicating that the Template Code Section (#CODE, #CONTROL, #EXTENSION, #PROCEDURE, #PROGRAM, or #MODULE) is valid.

The #RESTRICT structure contains Template language *statements* that evaluate the propriety of generating the section's source code. The #ACCEPT statement may be used to explicitly declare the section as appropriate. An implicit #ACCEPT also occurs if the #RESTRICT *statements* execute without encountering a #REJECT statement. The #REJECT statement must be used to specifically exclude the section from use. Both the #ACCEPT and #REJECT statements immediately terminate processing of the #RESTRICT code.

Example:

```
#CODE(ChangeControlSize,'Change control size')
  #WHERE(%EventHandling)
  #RESTRICT
    #CASE(%ControlType)
    #OF 'LIST'
    #OROF 'BUTTON'
      #REJECT
    #ELSE
      #ACCEPT
    #ENDCASE
  #ENDRESTRICT
  #PROMPT('Control to change',CONTROL),%MyField,REQ
  #PROMPT('New Width',@n04),%NewWidth
  #PROMPT('New Height',@n04),%NewHeight
%MyField{PROP:Width} = %NewWidth
%MyField{PROP:Height} = %NewHeight
```

See Also: #RESTRICT

#REJECT

#ADD (add to multi-valued symbol)

#ADD(*symbol, expression* [, *position*] **)**

#ADD	Adds a new instance to a multi-valued user-defined symbol.
symbol	A multi-valued user-defined symbol.
expression	An expression containing the value to place in the *symbol's* instance.
position	An integer constant or symbol containing the instance number to add to the *symbol*. Instance numbering begins with one (1). If the *position* is greater than the number of previously existing instances plus one, the new instance in appended and no intervening instances are instantiated.

The **#ADD** statement adds a value to a multi-valued user-defined *symbol*. An implied #FIX to that *symbol's* instance occurs. If the *symbol* is not a multi-valued user-defined symbol then a source generation error is produced.

If the *symbol* has been declared with the UNIQUE attribute, then the #ADD is a union operation into the existing set of *symbol's* values. Only one instance of the value being added may exist. Also, the UNIQUE attribute implies the #ADD is a sorted insert into the existing set of *symbol's* values. After each #ADD, all of the *symbol's* values will be in sorted order.

If the *symbol* has been declared without the UNIQUE attribute, duplicate values are allowed. The new value is added to the end of the list and may be a duplicate. If the *symbol* is a duplicate, then any dependent children instances are inherited.

Example:

```
#DECLARE(%ProcFilesPrefix),MULTI,UNIQUE   #!Declare unique multi-valued symbol
#FIX(%File,%Primary)                      #!Build list of all file prefiXEs in proc
#ADD(%ProcFilesPrefix,%FilePre)           #!Start with primary file
#FOR(%Secondary)                          #!Then add all secondary files
  #FIX(%File,%Secondary)
  #ADD(%ProcFilesPrefix,%FilePre)
#ENDFOR
```

See Also: #DECLARE

#ALIAS/#TRYALIAS (access a symbol from another instance)

#ALIAS(*newsymbol* , *oldsymbol* [, *instance*])

#TRYALIAS(*newsymbol* , *oldsymbol* [, *instance*])

#ALIAS / #TRYALIAS	Declares a synonym for a user-defined symbol.
newsymbol	Specifies the new synonym for the *oldsymbol*.
oldsymbol	The name of the symbol for which to declare a synonym. This must meet all the requirements of a user-defined symbol. This must not be a #PROMPT symbol or a variable in the same scope.
instance	An expression containing the instance of the addition containing the *oldsymbol*.

The **#ALIAS** statement declares a synonym for the user-defined *oldsymbol* declared in a #CODE, #CONTROL, or #EXTENSION template prompt for use in another.

The **#TRYALIAS** statement is similar to **#ALIAS** expect that the symbol is only defined by the template referred to by *oldsymbol*. If *oldsymbol* is not defined, no template error is posted.

Example:

```
#EXTENSION(GlobalSecurity,'Global Password Check'),APPLICATION
  #DECLARE(%PasswordFile)
  #DECLARE(%PasswordFileKey)

#EXTENSION(LocalSecurity,'Local Procedure Password Check'),PROCEDURE
  #ALIAS(%PswdFile,%PasswordFile,%ControlInstance)
  #ALIAS(%PswdFileKey,%PasswordFileKey,%ControlInstance)

#GROUP (%QueryHasLocator, %WhichInstance),AUTO
#TRYALIAS (%CurLocatorType, %LocatorType, %WhichInstance)
#IF (VAREXISTS(%CurLocatorType))
  #RETURN (%CurLocatorType)
#END
#RETURN ''
```

See Also:

#CODE

#CONTROL

#EXTENSION

#APPEND (add to source file)

#APPEND(*file* **)** [, **SECTION**]

#APPEND	Adds the *file* contents to the end of the current source output destination file.
file	A string constant, template symbol, or expression containing a DOS file specification. This may be a fully qualified DOS pathname.
SECTION	Specifies the error position of embedded source code is correctly patched to reflect the actual position in the generated code.

The **#APPEND** statement adds the complete contents of the *file* to the end of the current source output destination file. The contents of the *file* are NOT interpreted for source generation purposes. Therefore, the *file* should not contain any Template Language code.

If the *file* does not exist, #APPEND is ignored and source generation continues.

Example:

```
#FOR(%Module)
  #SET(%TempModuleFile,(%Module & '.$$$'))           #!Set temp module file
  #CREATE(%TempModuleFile)                           #!Create temp module file
  #FOR(%ModuleProcedure)                             #!For all procs in module
    #FIX(%Procedure,%ModuleProcedure)                #!Fix current procedure
    #GENERATE(%Procedure)                            #!Generate procedure code
  #ENDFOR                                            #!EndFor all procs in module
  #SET(%ModuleFile,(%Module & '.CLW'))               #!Set to current module file
  #CREATE(%ModuleFile)                               #!Create module file
  #GENERATE(%Module)                                 #!Generate module header
  #APPEND(%TempModuleFile),SECTION                   #!Add generated procedures
#ENDFOR
```

See Also: #SECTION

#APPLICATION (source generation control section)

#APPLICATION(*description* **)** [, **HLP(** *helpid* **)**] [**APPLICATION(** [*child(chain)*]

#APPLICATION	Begins source generation control section.
description	A string constant describing the application section.
HLP	Specifies on-line help is available.
helpid	A string constant containing the identifier to access the Help system. This may be either a Help keyword or "context string."
APPLICATION	Tells the Application Generator to automatically place the named) *child* template on every procedure.
child(chain)	The name of a #EXTENSION with the PROCEDURE attribute to automatically populate into every generated procedure when the #EXTENSION with the APPLICATION attribute is populated.

The **#APPLICATION** statement marks the beginning of a source generation control section. The section is terminated by the next Template Code Section (#PROGRAM, #MODULE, #PROCEDURE, #CONTROL, #CODE, #EXTENSION, #UTILITY, or #GROUP) statement. The Template statements contained in this section control the source generation process. Only one #APPLICATION section is allowed in a single Template set. Actual source generation is done by the #GENERATE statement.

Any User-defined symbols defined in the #APPLICATION section are available for use in any Template Code Section that is generated. Any prompts in this section are placed on the Global Properties window and have global scope.

Example:

```
#APPLICATION('Example Application Section')       #!Generate entire application
#PROMPT('Enable &Shared Files',CHECK),%SharedFiles
#PROMPT('Close Unused &Files',CHECK),%CloseFiles,DEFAULT(1)
#BUTTON('.INI File Settings')
  #PROMPT('Use .INI file',CHECK),%INIActive,DEFAULT(1)
  #ENABLE(%INIActive)
    #PROMPT('.INI File to use',DROP,'Program Name.INI|Other'),%INIFile
    #ENABLE(%INIFile='Other')
      #PROMPT('File Name',@S40),%ININame
    #ENDENABLE
    #PROMPT('Save Window Locations',CHECK),%INISaveWindow,DEFAULT(1)
  #ENDENABLE
#ENDBUTTON
#!
#!Global Template Declarations.
```

```
#MESSAGE('Generating ' & %Application,0) #! Open the Message Box
#DECLARE(%FilesUsed),UNIQUE,MULTI                 #! Label of every file used
#DECLARE(%FilePut,%FilesUsed)                     #! "Yes" for RI PUT used
#DECLARE(%FileDelete,%FilesUsed)                  #! "Yes" for RI DELETE used
#DECLARE(%ModuleFilesUsed,%Module),UNIQUE,MULTI,SAVE  #!Name of file used in module
#DECLARE(%ModuleFilePut,%ModuleFilesUsed),SAVE    #! "Yes" for RI PUT used
#DECLARE(%ModuleFileDelete,%ModuleFilesUsed),SAVE #! "Yes" for RI DELETE used
#DECLARE(%IniFileName)                            #! Used to construct INI file
#DECLARE(%ModuleProcs,%Module),MULTI,SAVE,UNIQUE  #! Program MAP prototype
#DECLARE(%ModulePrototype,%ModuleProcs)           #! Module MAP prototype
#DECLARE(%AccessMode)                             #! File open mode equate
#DECLARE(%BuildFile)                              #! Construction filename
#!
#!Initialization Code for Global User-defined Symbols.
#IF(%SharedFiles)                                 #! IF Shared Files Enabled
  #SET(%AccessMode,'42h')                         #! default access 'shared'
#ELSE                                             #! ELSE (IF NOT Shared Files ..)
  #SET(%AccessMode,'22h')                         #! default access 'open'
#ENDIF                                            #! END (IF Shared Files ...)
#IF(%INIFile = 'Program Name.INI')                #! IF using program.ini
  #SET(%INIFileName, %Application & '.INI')       #! SET the file name
#ELSE                                             #! ELSE (IF NOT using Program.ini)
  #SET(%INIFileName,%ININame)                     #! SET the file name
#ENDIF                                            #! END (IF using program.ini)
#!
#! Main Source Code Generation Loop.
#DECLARE(%GlobalRegenerate)                       #! Flag that controls generation
#IF(~%ConditionalGenerate OR %DictionaryChanged OR %RegistryChanged)
  #SET(%GlobalRegenerate,%True)                   #! Generate Everything
#ELSE                                             #! ELSE (If no global change)
  #SET(%GlobalRegenerate,%False)                  #! Generate changed modules only
#ENDIF                                            #! END (IF Global Change)
#SET(%BuildFile,(%Application & '.TM$'))          #! Make temp program filename
#FOR(%Module), WHERE (%Module <> %Program)        #! For all member modules
  #MESSAGE('Generating Module:    ' & %Module, 1) #! Post generation message
  #IF(%ModuleChanged OR %GlobalRegenerate)        #! IF module to be generated
    #FREE(%ModuleProcs)                           #! Clear module prototypes
    #FREE(%ModuleFilesUsed)                       #! Clear files used
    #CREATE(%BuildFile)                           #! Create temp module file
    #FOR(%ModuleProcedure)                        #! FOR all procs in module
      #FIX(%Procedure,%ModuleProcedure)           #! Fix current procedure
      #MESSAGE('Generating Procedure: ' & %Procedure, 2) #! Post generation message
      #GENERATE(%Procedure)                       #! Generate procedure code
    #ENDFOR                                       #! END (For all procs in module)
    #CLOSE(%BuildFile)                            #! Close last temp file
    #CREATE(%Module)                              #! Create a module file
    #GENERATE(%Module)                            #! Generate module header
    #APPEND(%BuildFile)                           #! Append the temp mod file
    #CLOSE(%Module)                                  #! Close the module file
  #ENDIF                                             #! END (If module to be...)
#ENDFOR                                              #! END (For all member modules)
#FIX(%Module,%Program)                               #! FIX to program module
#MESSAGE('Generating Module:    ' & %Module, 1)      #! Post generation message
#FREE(%ModuleProcs)                                  #! Clear module prototypes
#FREE(%ModuleFilesUsed)                              #! Clear files used
#CREATE(%BuildFile)                                  #! Create temp module file
#FOR(%ModuleProcedure)                               #! For all procs in module
  #FIX(%Procedure,%ModuleProcedure)                  #! Fix current procedure
  #MESSAGE('Generating Procedure: ' & %Procedure, 2) #! Post generation message
  #GENERATE(%Procedure)                              #! Generate procedure code
#ENDFOR                                              #! EndFor all procs in module
#CLOSE()                                             #! Close last temp file
```

See Also: #GENERATE

#ASSERT (evaluate assumption)

#ASSERT(*condition* [, *message*] **)**

#ASSERT	Evaluates assumptions to detect unexpected errors.
condition	An expression that should always evaluate as true.
message	A string constant containing the text to display if the *condition* is not true. If omitted, the text of the *condition* is displayed.

#ASSERT evaluates assumptions to detect unexpected errors. #ASSERT eveluates the *condition* expression and displays the *message* if the *condition* is not true. This allows you to test the veracity of any programming assumptions you are making at any point in your template code.

Example:

```
#ASSERT(%False)                   #!Displays '%False'
#ASSERT(%False, 'Message Text')   #!Displays 'Message Text'
```

#AT (insert code in an embed point)

#AT(*location*[, *instances*] **)** [, **WHERE(***expression***)**] [, **AUTO**] [, **PRESERVE**] [, **DESCRIPTION**(*text*)]
[, **PRIORITY(** *number* **)**] [, **FIRST**] [, **LAST**]
statements

#ENDAT

#AT	Specifies a *location* to generate *statements*.
location	An #EMBED *identifier.* This may be a #EMBED for a procedure that comes from another template set.
instances	The *location* parameters that identify the embedded source code point for a multi-valued #EMBED *identifier*. There may as many *instance* parameters as are required to explicitly identify the embedded source code point. These *instances* are omittable.
WHERE	More closely specifies the #AT *location* as only those embed points where the *expression* is true.
expression	An expression that specifies exact placement.
AUTO	Opens a new scope for the #AT. This means that any #DECLARE statements in the #AT would not be available to the #PROCEDURE being generated.
PRESERVE	Preserves the current fixed instances of all built-in multi-valued symbols when the #AT is called and restores all those instances when the #AT code terminates.
DESCRIPTION	Specifies *text* that will be displayed at the embed tree node specified by the *location*.
PRIORITY	Specifies the order inwhich the #AT is generated into the *location*. The lowest value generates first.
number	An integer constant in the range 1 to 10000.
FIRST	Equivalent to PRIORITY(1).
LAST	Equivalent to PRIORITY(10000).
statements	Template and/or target language code.
#ENDAT	Terminates the section.

The **#AT** structure specifies a *location* to generate *statements*. #AT is valid only in a #CONTROL, #CODE, or #EXTENSION templates, and is used to allow them to generate *statements* into multiple *locations*. The #AT structure must terminate with **#ENDAT**.

The WHERE clause allows you to create an *expression* that can specify a single specific instance of a #EMBED that has a *symbol* attribute. You may not place #AT within any type of conditional structure (such as #IF or #CASE). If you need to conditionally generate the code, place the #IF or #CASE structure within the #AT structure.

Example:

```
#CONTROL(BrowseList,'Add Browse List controls')
  #AT(%ControlEvent,'?Insert','Accepted'),PRIORITY(5000)
    #IF(%InsertAllowed)
GlobalRequest = InsertRecord
%UpdateProc
    #ENDIF
  #ENDAT
#!
```

See Also: #EMBED

#CODE

#CONTROL

#EXTENSION

#RESTRICT

#ATEND (template reset code)

#ATEND

statements

#ENDAT

#ATEND	Specifies template code to execute after the #PROCEDURE, #CODE, #CONTROL, or #EXTENSION generates.
statements	Template language code.
#ENDAT	Terminates the section.

The **#ATEND** structure specifies template code to execute after the #PROCEDURE, #CODE, #CONTROL, or #EXTENSION generates its code. Therefore, the *statements* should only contain Template language. #ATEND is usually used to reset internal template variables. You may not place #ATEND within any type of conditional structure (such as #IF or #CASE). If you need to conditionally generate the code, place the #IF or #CASE structure within the #ATEND structure.

Example:

```
#CONTROL(BrowseList,'Add Browse List controls')
 #ATEND
  #SET(%ListQueue,%NULL)
 #ENDAT
```

See Also:	#PROCEDURE
	#CODE
	#CONTROL
	#EXTENSION

#ATSTART (template intialization code)

#ATSTART

statements

#ENDAT

#ATSTART	Specifies template code to execute before the #PROCEDURE, #CODE, #CONTROL, or #EXTENSION generates.
statements	Template language code.
#ENDAT	Terminates the section.

The **#ATSTART** structure specifies template code to execute before the #PROCEDURE, #CODE, #CONTROL, or #EXTENSION generates its code. Therefore, the *statements* should normally only contain Template language. #ATSTART is usually used to initialize internal template variables. You may not place #ATSTART within any type of conditional structure (such as #IF or #CASE). If you need to conditionally generate the code, place the #IF or #CASE structure within the #ATSTART structure.

Example:

```
#CONTROL(BrowseList,'Add Browse List controls')
 #ATSTART
  #FIX(%Control,%ListBox)
  #DECLARE(%ListPre)
  #SET(%ListPre,'List' & %ActiveTemplateInstance & ':')
                      #!Makes %ListPre contain "List#:'
 #ENDAT
 #AT(%DataSectionBeforeWindow)
%ListPre:Scroll  LONG      !Scroll for %Control
%ListPre:Chioce  LONG      !Choice for %Control
 #ENDAT
```

See Also:	#PROCEDURE
	#CODE
	#CONTROL
	#EXTENSION

#BOXED (prompt group box)

#BOXED([*string*] **)** [, **AT( )**] [, **WHERE(** *expression* **)**] [, **CLEAR**] [, **HIDE**] [, **SECTION**]

prompts

#ENDBOXED

#BOXED	Creates a group box of *prompts*.
string	A string constant containing the text to display as the group box caption.
AT	Specifies the position of the group in the window, relative to the first prompt placed on the window from the Template (excluding the standard prompts on every procedure properties window). This attribute takes the same parameters as the Clarion language AT attribute.
WHERE	Specifies the #BOXED is visible only for those instances where the *expression* is true.
expression	An expression that specifies the condition for use.
CLEAR	Specifies the *prompts* symbol values are cleared when disabled.
HIDE	Specifies the *prompts* are hidden if the WHERE *expression* is false when the dialog is first displayed. If no WHERE attribute is present, *prompts* are always hidden.
SECTION	Specifies all AT() attributes for the *prompts* are positioned relative to the start of the #BOXED section.
prompts	One or more #PROMPT statements. This may also contain #DISPLAY, #VALIDATE, #ENABLE, #PREPARE, and #BUTTON statements.
#**ENDBOXED**	Terminates the group box of *prompts*.

The **#BOXED** statement creates a group box displaying the *string* as its caption. If the WHERE attribute is present, the *prompts* are hidden or visible based upon the evaluation of the *expression*. If the *expression* is true, the *prompts* are visible, otherwise they are hidden.

Example:

```
#PROMPT('Pick One',OPTION),%InputChoice        #!These prompts on second page
#PROMPT('Choice One',RADIO)
#PROMPT('Choice Two',RADIO)
#BOXED('Choice Two Options'),WHERE(%InputChoice = 'Choice Two')
    #PROMPT('Screen Field',CONTROL),%SomeField
    #VALIDATE(%ScreenFieldType = 'LIST','Must select a list box')
#ENDBOXED
```

See Also: #PROMPT, #VALIDATE

#BREAK (break out of a loop)

#BREAK

The **#BREAK** statement immediately breaks out of the #FOR or #LOOP structure in which it is enclosed. Control passes to the next statement following the #ENDFOR or #ENDLOOP. #BREAK is only valid within a #FOR or #LOOP structure, else an error is generated during Template file pre-processing. #BREAK acts as a #RETURN statement if issued from within a #GROUP inserted in the loop (unless it is within a #FOR or #LOOP structure completely contained within the #GROUP).

Example:

```
#SET(%StopFile,'CUSTOMER')
#FOR(%File)
  #IF (UPPER(%File) = %StopFile)
    #BREAK
  #ENDIF
  OPEN(%File)
#ENDFOR
```

#BUTTON (call another page of prompts)

```
#BUTTON( string [, icon ] ) [, HLP( id ) ] [, AT() ] [, REQ ] [, INLINE] [, WHENACCEPTED(group) ]
                      [,        | FROM( fromsymbol, expression ) [, WHERE( condition ) ]      | ]
                                | MULTI( multisymbol, expression )  |

  prompts

#ENDBUTTON
```

#BUTTON	Creates a command button to call another page of *prompts*.
string	An expression containing the text to display on the button's face. This may contain an ampersand (&) to indicate the "hot" letter for the button.
icon	A string constant containing the name of an .ICO file or standard icon to display on the button's face. The *string* then serves only for "hot" key definition.
HLP	Specifies on-line help is available for the #BUTTON.
id	A string constant containing the identifier to access the Help system. This may be either a Help keyword or "context string."
AT	Specifies the position of the button in the window, relative to the first prompt placed on the window from the Template (excluding the standard prompts on every procedure properties window). This attribute takes the same parameters as the Clarion language AT attribute. Specifying zero width and height indicates the #BUTTON occupies no space on the prompt dialog.
REQ	Specifies the programmer must press the button at least once when the procedure is created.
FROM	Specifies the programmer may enter a set of values for the *prompts* for each instance of the *fromsymbol*.
fromsymbol	A built-in multi-valued symbol which pre-defines all instances on which the *prompts* symbols are dependent. The programmer may not add, change, or delete instances of the *fromsymbol*.
expression	A string expression to format data display in the multiple value display list box.
WHERE	Specifies the #BUTTON displays only those instances of the *fromsymbol* where the *condition* is true.
condition	An expression that specifies the condition for use.
MULTI	Specifies the programmer may enter multiple sets of values for the prompts. This allows multiple related groups of prompts.
multisymbol	A user-defined symbol on which all the *prompts* symbols are dependent. This symbol is internally assigned a unique value for each set of *prompts*.

INLINE	The multiple values the programmer enters for the #BUTTON appears as a list box with update buttons which allow the programmer to enter multiple values. The MULTI or FROM attribute must also be present.
WHENACCEPTED	Specifies the named *group* to execute when the #BUTTON is pressed.
group	The name of the #GROUP to execute.
prompts	One or more #PROMPT statements. This may also contain #DISPLAY, #VALIDATE, #ENABLE, and #BUTTON statements.
#ENDBUTTON	Terminates the group of *prompts* which are on the page called by the #BUTTON.

The **#BUTTON** statement creates a command button displaying either the *string* or the *icon* on its face. When the programmer presses the button, a new page of *prompts* appears for selection and entry.

Each new page of *prompts* has its own OK, CANCEL, and TEMPLATE HELP buttons as standard fields. All other fields on the page are generated from the *prompts* within the #BUTTON structure.

Each page's OK button closes the current page of *prompts*, saving the data the programmer entered in the *prompts*, then returns to the prior window. The CANCEL button closes the current page of *prompts* without saving, then returns to the prior window. If the page calls another page with a nested #BUTTON statement and the programmer presses OK on the lowest level page, then CANCEL on the page that called it, the entire transaction is cancelled.

The MULTI attribute specifies the programmer may enter multiple sets of values for the *prompts*. The button calls a window containing a list box to display all the multiple values entered for the sets of *prompts*, along with Insert, Change, and Delete buttons. These three buttons call another window containing all the *prompts* to allow the programmer to update the entries in the list. The *expression* is used to format the information for display in the list box.

The FROM attribute also specifies the programmer may enter multiple sets of values for the *prompts*. The button calls a window containing a list box that displays each instance of the *fromsymbol*, along with an Edit button. This button calls another window containing all the *prompts* to allow the programmer to update the entries associated with that instance of the *fromsymbol*. The *expression* is used to format the information for display in the list box. The WHERE attribute may be used to limit the instances of the *fromsymbol* to only those that meet the WHERE *condition*.

Example:

```
#PROMPT('Name a File',FILE),%FileName           #!Prompt on the first page

#BUTTON('Page Two')                             #!Button on first page calls
  #PROMPT('Pick One',OPTION),%InputChoice       #!These prompts on second page
  #PROMPT('Choice One',RADIO)
  #PROMPT('Choice Two',RADIO)
  #ENABLE(%InputChoice = 'Choice Two')
    #PROMPT('Screen Field',CONTROL),%SomeField
    #VALIDATE(%ScreenFieldType = 'LIST','Must select a list box')
  #ENDENABLE
#ENDBUTTON                                      #!Terminate second page prompts

#PROMPT('Enter some value',@S20),%InputValue1    #!Another prompt on first page

#!Multiple input button:
#BUTTON('Multiple Names'),MULTI(%ButtonSymbol,%ForeName & ' ' & %SurName)
  #PROMPT('First Name',@S20),%ForeName
  #PROMPT('Last Name',@S20),%SurName
#ENDBUTTON                                      #!Terminate second page prompts

#PROMPT('Enter another value',@S20),%InputValue2 #!Another prompt on first page
 #!Multiple input button dependent on %File:
#BUTTON('File Options'),FROM(%File)
  #PROMPT('Open Access Mode',DROP('Open|Share'),%FileOpenMode
#ENDBUTTON                                      #!Terminate second page prompts
```

See Also:

#PROMPT

#VALIDATE

#ENABLE

#CALL (insert code from a #GROUP, without indention)

#CALL(*symbol* [**(** *set* **)**] [, *parameters*] **)** [, *returnvalue*]

#CALL	Inserts code from a #GROUP, retaining the indention in its #GROUP.
symbol	A symbol that names a #GROUP section.
set	The #TEMPLATE *name* parameter for the template set to which the #GROUP belongs. If omitted, the #GROUP must be of the same template set *name* as the #PROCEDURE in which it is used.
parameters	The parameters passed to the #GROUP. Each parameter must be separated by a comma. All parameters defined for the #GROUP must be passed; they may not be omitted.
returnvalue	A symbol to receive the value returned by the #RETURN statement.

The **#CALL** statement places the code from the #GROUP named by the *symbol*, retaining the indention in its #GROUP--any source in column 1 in the #GROUP is generated into column 1, no matter where the #CALL is placed. This is equivalent to using #INSERT with the NOINDENT parameter.

The *set* parameter specifies the #TEMPLATE that contains the #GROUP. This allows any Template to use #GROUP code from any other registered Template. The *parameters* passed to the #GROUP fall into two categories: value-parameters and variable-parameters. Value-parameters are declared by the #GROUP as a user-defined symbol, while variable-parameters are declared by the #GROUP as a user-defined symbol with a prepended asterisk (*). Either a symbol or an expression may be passed as a value-parameter. Only a symbol may be passed as a variable-parameter.

The *returnvalue* symbol receives the value returned by the #GROUP from the #RETURN statement that terminates the #GROUP. If the #GROUP does not contain a #RETURN statement, or that #RETURN does not have a parameter, then the value received is an empty string ('').

Example:

```
#CALL(%SomeGroup)                      #!Ordinary insert retaining original indention
#CALL(%GenerateFormulas(Clarion))      #!Insert #GROUP from Clarion Template
#CALL(%FileRecordFilter,%Secondary)    #!Insert #GROUP with passed parameter
#!#GROUP from Clarion Template with two passed parameters
#CALL(%FileRecordFilter(Clarion),%Primary,%Secondary)
```

See Also: #GROUP, #INSERT, #INVOKE, #RETURN

#CASE (conditional execution structure)

#CASE(*condition* **)**
#OF(*expression* **)**
[**#OROF(** *expression* **)**]
statements
[**#ELSE**
statements]
#ENDCASE

#CASE	Initiates a selective execution structure.
condition	Any Template Language expression which returns a value.
#OF	The #OF *statements* are executed when the #OF *expression* is equal to the *condition* of the CASE. There may be many #OF options in a #CASE structure.
expression	Any Template Language expression which returns a value.
#OROF	The #OROF *statements* are executed when the #OROF *expression* is equal to the *condition* of the #CASE. There may be many #OROF options associated with one #OF option.
#ELSE	The #ELSE *statements* are executed when all preceding #OF and #OROF *expressions* are not equal to the *condition* of the #CASE. #ELSE (if used) must be the last option in the #CASE structure.
statements	Any valid executable source code.
#ENDCASE	Terminates the #CASE structure.

A **#CASE** structure selectively executes *statements* based on equivalence between the #CASE *condition* and one of the #OF or #OROF *expressions*. If there is no exact match, the *statements* following #ELSE are executed. The #CASE structure must be terminated by **#ENDCASE**. If there is no #ENDCASE, an error message is issued during Template file pre-processing. #CASE structures may be nested within other #CASE structures.

Example:

```
#CASE(%ScreenField)
#OF('?Ok')
  #INSERT(%OkButtonGroup)
#OF('?Cancel')
#OROF('?Exit')
  #INSERT(%CancelButtonGroup)
#ELSE
  #INSERT(%OtherControlsGroup)
#ENDCASE
```

#CLASS (define a formula class)

#CLASS(*string, description* **)**

#CLASS	Defines a formula class.
string	A string constant containing the formula class.
description	A string expression containing the description of the formula class to display in the list of those available in the Formula Editor.

The **#CLASS** statement defines a formula class for use in the Formula Editor. The Formula Class allows the Template to determine the precise logical position at which the formula appears in the generated source code.

Example:

```
#PROCEDURE(SomeProc,'An Example Template'),WINDOW
#CLASS('START','At beginning of procedure')
#CLASS('LOOP','In process loop')
#CLASS('END','At end of procedure')
%Procedure PROCEDURE
%ScreenStructure
  CODE
  #INSERT(%GenerateFormulas,'START')      #!Generate START class formulas
  OPEN(%Screen)
  ACCEPT
    #INSERT(%GenerateFormulas,'LOOP')     #!Generate LOOP class formulas
  END
  #INSERT(%GenerateFormulas,'END')        #!Generate END class formulas
```

#CLEAR (clear single-valued symbol)

#CLEAR(*symbol* **)**

#CLEAR	Removes the value from a single-valued user-defined symbol.
symbol	A single-valued user-defined symbol.

The **#CLEAR** statement removes the value from a single-valued user-defined symbol. This statement is approximately the same as using #SET to assign a null value to the *symbol*, except it is more efficient.

Example:

```
#DECLARE(%SomeSymbol)                 #!Declare symbol
#SET(%SomeSymbol,'Value')             #!Assign a value
                                      #!%SomeSymbol now contains: 'Value'
#CLEAR(%SomeSymbol)                   #!Clear value
                                      #!%SomeSymbol now contains: ''
```

See Also: #DECLARE

#ADD

#CLOSE (close source file)

#CLOSE([*file*] **)** [, **READ**]

#CLOSE	Closes an open generated source code disk file.
file	A string constant, template symbol, or expression containing a DOS file specification. This may be a fully qualified DOS pathname. If omitted, the current disk file receiving generated source code is closed.
READ	Closes the read-only input file.

The **#CLOSE** statement closes an open disk file receiving the generated source code. If the *file* is omitted, the current disk file receiving generated source code is closed. If the *file* does not exist, or is already closed, a source generation error is produced.

Example:

```
#SET(%NewProgramFile,(%Application & '.$$$'))    #!Temp new program filename
#CREATE(%NewProgramFile)                         #!Create new program file
#GENERATE(%Program)                              #!Generate main program header
#CLOSE(%NewProgramFile)                          #!Create new program file

#OPEN(%ProgramFile),READ                         #!Open it in read-only mode
#DECLARE(%ASCIIFileRecord)
#LOOP
 #READ(%ASCIIFileRecord)
 #! Parse the line and do something with it
 #IF(%ASCIIFileRecord = %EOF)
  #BREAK
 #ENDIF
#ENDLOOP
#CLOSE(%ProgramFile),READ                        #!Close the read-only file
```

See Also:	#OPEN
	#READ

#CODE (define a code template)

```
#CODE( name,description [,target ] )[, SINGLE][, HLP( helpid )] [, PRIMARY( message [, flag] )]
                    [, DESCRIPTION( expression ) ] [, ROUTINE ] [, PRIORITY( number )]
                    [, REQ( addition [,| BEFORE      | ] ) ] [,   | FIRST  | ]
                                       | AFTER |                  | LAST   |
```

#CODE	Begins a code template that generates source into an embedded source code point.
name	The label of the code template. This must be a valid Clarion label.
description	A string constant describing the code template.
target	A string constant that specifies the source language the code template generates. If omitted, it defaults to Clarion. This restricts the #CODE to matching *target* language use, only.
SINGLE	Specifies the #CODE may be used only once in a given procedure (or program, if the embedded source code point is global).
HLP	Specifies on-line help is available.
helpid	A string constant containing the identifier to access the Help system. This may be either a Help keyword or "context string."
PRIMARY	Specifies a primary file for the code template must be placed in the procedure's File Schematic.
message	A string constant containing a message that appears in the File Schematic next to the #CODE's Primary file.
flag	Either OPTIONAL (the file is not required), OPTKEY (the key is not required), or NOKEY (the file is not required to have a key).
DESCRIPTION	Specifies the display description of a #CODE that may be used multiple times in a given application or procedure.
expression	A string constant or expression that contains the description to display.
ROUTINE	Specifies the generated code is not automatically indented from column one.
PRIORITY	Specifies the order inwhich the #CODE is generated into the embed point. The lowest value generates first.
number	An integer constant in the range 1 to 10000.
REQ	Specifies the #CODE requires a previously placed #CODE, #CONTROL, or #EXTENSION before it may be used. It also means all prompts and variables of the required *addition* are available to it.

addition	The name of the previously placed #CODE, #CONTROL, or #EXTENSION template, from any template set.
BEFORE	Legacy attribute, replaced by PRIORITY.
AFTER	Legacy attribute, replaced by PRIORITY.
FIRST	Equivalent to PRIORITY(1).
LAST	Equivalent to PRIORITY(10000).

#CODE defines the beginning of a code template which can generate code into embedded source code points. A #CODE section may contain Template and/or target language code. The #CODE section is terminated by the first occurrence of a Template Code Section (#PROGRAM, #MODULE, #PROCEDURE, #CONTROL, #CODE, #EXTENSION, or #GROUP) statement, or the end of the file. Within a single Template set, separate #CODE sections may not be defined with the same *name*.

#CODE generates its code into a #EMBED embedded source code point. The generated code is automatically indented when placed in ROUTINEs, unless the ROUTINE attribute is present. A #CODE section may contain #PROMPT statements to prompt for the values needed to generate proper source code. It may also contain #EMBED statements, which become active only if the #CODE section is used.

You can use the #WHERE statement to limit the availability of the #CODE to those embedded source code points where the generated code would be appropriate. A #CODE may contain multiple #WHERE statements to explicitly define all the valid embedded source code points in which it may appear. #RESTRICT can also further restrict the availability of the #CODE based on an expression or Template language statements.

The #AT/#ENDAT structure allows a single #CODE to generate code into multiple embedded source code points to support its functionality.

Example:

```
#CODE(ChangeProperty,'Change control property')
  #WHERE(%SetupWindow..%ProcedureRoutines)   #!Appropriate only after window open
  #PROMPT('Control to change',CONTROL),%MyField,REQ
  #PROMPT('Property to change',@S20),%Property,REQ
  #PROMPT('New Value',@S20),%Value,REQ
%MyField{%Property} = '%Value'               #<!Change the %Property of %MyField
```

See Also: #EMBED

#WHERE

#RESTRICT

#AT (insert code in an embed point)

#COMMENT (specify comment column)

#COMMENT(*column* **)**

#COMMENT	Sets the default column number for aligned comments.
column	A numeric constant in the range 1 to 255.

The **#COMMENT** statement sets the default column number in which Clarion comments prefaced with the #<! statement will be generated by the Application Generator.

Example:

```
#COMMENT(50)     #!Set comment column
IF Action = 1    #<!If adding a record
  SomeVariable = InitVariable
END
```

See Also: #<

#CONTEXT (set template code generation context)

#CONTEXT(*section* [, *instance*] **)**

statements

#ENDCONTEXT

#CONTEXT	Sets the code generation context to emulate generating the named *section*.
section	One of the following valid symbols: %Application, %Procedure, %Module, or %Program.
instance	A symbol containing a valid instance number of a code, control ,or extension template used in the named *section*.
statements	Template language code.
#ENDCONTEXT	Terminates the context change.

The **#CONTEXT** structure specifies template code that executes as if the source code for the named *section* were being generated. Once the context has been set, all the prompt symbols , declared symbols, and embeds in the named *section* are in scope and available for use in the #CONTEXT *statements*. If an *instance* is specified, the prompts for the corresponding component are brought into scope. #CONTEXT is valid for use anywhere in template code.

The key to understanding the use of the #CONTEXT structure is "**as if the source code for the named *section* were being generated**." This statement means that the *statements* are evaluated as if #GENERATE were executing. For example, a #EMBED statement within a #CONTEXT structure does not name a new embed point but instead, generates the contents of the named embed point.

Example:

```
#CODE(StealSomeCode,'Preparing to Process the Window Code Stealer')
#PROMPT('Procedure Name',FROM(%Procedure)),%ProcToStealFrom,REQ
#FIX(%Procedure,%ProcToStealFrom)
#CONTEXT(%Procedure)
 #EMBED(%BeforeAccept,'Preparing to Process the Window')
#ENDCONTEXT
```

See Also:	#CODE
	#CONTROL
	#EXTENSION

#CONTROL (define a control template)

```
#CONTROL( name, description ) [, MULTI ] [, PRIMARY( message [, flag ] ) ] [, SHOW ]
        [, WINDOW ] [, REPORT] [, WRAP( control ) ] [, PRIORITY( number )]
        [, REQ( addition [,| BEFORE    | ] ) [,   | FIRST | ] [,DESCRIPTION( expression ) ] ]
                           | AFTER|               | LAST  |

        CONTROLS

            control statements [, #REQ ]

        END
```

#CONTROL	Begins a code template that generates a set of controls into a window and the source code required to manipulate them into embedded source code points.
name	The label of the template (must be a valid Clarion label).
description	A string constant describing the control template.
MULTI	Specifies the #CONTROL may be used multiple times in a given window.
PRIMARY	Specifies a primary file for the set of controls must be placed in the procedure's File Schematic.
message	A string constant or template symbol containing a message that appears in the File Schematic next to the #CONTROL's Primary file.
flag	Either OPTIONAL (the file is not required), OPTKEY (the key is not required), or NOKEY (the file is not required to have a key).
SHOW	Specifies the #CONTROL prompts are placed on the procedure properties window.
WINDOW	Tells the Application Generator to make the #CONTROL available in the Window Formatter. This is the default setting if both WINDOW and REPORT are omitted.
REPORT	Tells the Application Generator to make the #CONTROL available in the Report Formatter. If omitted, the #CONTROL may not be placed in a REPORT.
WRAP	Specifies the #CONTROL template is offered as an option for the *control* when the "Translate controls to control templates when populating" option is set in Application Options.
control	The data type of the control for which the #CONTROL is a viable alternative.
PRIORITY	Specifies the order inwhich the #CONTROL is generated. The lowest value generates first.
number	An integer constant in the range 1 to 10000.

REQ	Specifies the #CONTROL requires a previously placed #CODE, #CONTROL, or #EXTENSION before it may be used.
addition	The name of the previously placed #CODE, #CONTROL, or #EXTENSION.
BEFORE	Legacy attribute, replaced by PRIORITY.
AFTER	Legacy attribute, replaced by PRIORITY.
FIRST	Equivalent to PRIORITY(1).
LAST	Equivalent to PRIORITY(10000).
DESCRIPTION	Specifies the display description of a #CONTROL that may be used multiple times in a given application or procedure.
expression	A string constant or expression that contains the description to display.
CONTROLS	Specifies the *controls* for the #CONTROL, and must be terminated with an END statement. This is a "pseudo-Clarion keyword" in that, if you replace the CONTROLS statement with a WINDOW statement, you can use the Text Editor's Window Formatter to create the *controls*.
controls	Window control declarations that specifiy the control set belonging to the #CONTROL.
#REQ	Specifies the *control* is required. If deleted from the window or report, the entire #CONTROL (including all its *controls*) is deleted.

#CONTROL defines the beginning of a code template containing a "matched set" of controls to populate into a window or report as a group. It also generates the source code required for their correct operation into embedded source code points. A #CONTROL section may contain Template and/or target language code. The #CONTROL section is terminated by the first occurrence of a Template Code Section (#PROGRAM, #MODULE, #PROCEDURE, #CONTROL, #CODE, #EXTENSION, or #GROUP) statement, or the end of the file. Within a single Template set, separate #CONTROL sections may not be defined with the same *name*.

#CONTROL generates the code to operate its *controls* into #EMBED embedded source code points using the #AT/#ENDAT structure. #RESTRICT can restrict use of the #CONTROL based on an expression or Template language statements.

A #CONTROL section may contain #PROMPT statements to prompt for the values needed to generate proper source code. These prompts appear on the Actions window in the environment. It may also contain #EMBED statements which become active only if the #CONTROL section is used.

The *x* and *y* parameters of the AT attribute of the *controls* in the #CONTROL set determine the positioning of the *control* relative to the last control in the #CONTROL set placed on screen (or relative to the window, if first). If these parameters are omitted, the programmer is prompted for the position to place the *control.* This makes it simple to populate an entire set of *controls* without requiring the programmer to place each one individually.

The WRAP attribute specifies the #CONTROL is offered as an option to the programmer when the "Translate controls to control templates when populating" option is set in Application Options. The control parameter specifies the type of control to which the #CONTROL applies. This makes the #CONTROL a "wrapper" for the control type, such that, when the programmer populates the control in the formatter, a dialog appears offering the choice of populating either the control itself, or the #CONTROL template. For example, with the WRAP(LIST) attribute on a #CONTROL, when the programmer attempts to populate a LIST control a dialog appears offering the opportunity to use either the #CONTROL template (which generates executable code to "drive" the control) or the LIST control itself (requiring the programmer to write the "driving" code for the control).

Example:

```
#CONTROL(BrowseList,'Add Browse List controls')
  #PROMPT('Allow Inserts',CHECK),%InsertAllowed,DEFAULT(1)
  #ENABLE(%InsertAllowed)
      #PROMPT('Insert Hot Key',@s20),%InsertHotKey,DEFAULT('InsertKey')
  #ENDENABLE
  #PROMPT('Allow Changes',CHECK),%ChangeAllowed,DEFAULT(1)
  #ENABLE(%ChangeAllowed)
      #PROMPT('Change Hot Key',@s20),%ChangeHotKey,DEFAULT('CtrlEnter')
  #ENDENABLE
  #PROMPT('Allow Deletes',CHECK),%DeleteAllowed,DEFAULT(1)
  #ENABLE(%DeleteAllowed)
      #PROMPT('Delete Hot Key',@s20),%DeleteHotKey,DEFAULT('DeleteKey')
  #ENDENABLE
  #PROMPT('Update Procedure',PROCEDURE),%UpdateProc
  CONTROLS
   LIST,AT(,,270,99),USE(?List),IMM,FROM(Queue:Browse),#REQ
   BUTTON('Insert'),AT(,,40,15),USE(?Insert),MSG('Add record')
   BUTTON('Change'),AT(,,40,15),USE(?Change),DEFAULT,MSG('Change Record')
   BUTTON('Delete'),AT(,,40,15),USE(?Delete),MSG('Delete record')
  END
#!
  #AT(BeforeAccept)
    #IF(%InsertAllowed)
?Insert{PROP:Key} = %InsertHotKey
    #ENDIF
    #IF(%ChangeAllowed)
?Change{PROP:Key} = %ChangeHotKey
    #ENDIF
    #IF(%DeleteAllowed)
?Delete{PROP:Key} = %DeleteHotKey
    #ENDIF
  #ENDAT
#!
  #AT(%ControlEvent),WHERE(%ControlOriginal='?Insert' AND %ControlEvent='Accepted')
    #IF(%InsertAllowed)
```

```
Action = AddRecord
%UpdateProc
    #ENDIF
  #ENDAT
#!
  #AT(%ControlEvent),WHERE(%ControlOriginal='?Chg' AND %ControlEvent='Accepted')
    #IF(%ChangeAllowed)
Action = ChangeRecord
%UpdateProc
    #ENDIF
  #ENDAT
#!
  #AT(%ControlEvent),WHERE(%ControlOriginal='?Delete' AND %ControlEvent='Accepted')
    #IF(%DeleteAllowed)
Action = DeleteRecord
%UpdateProc
    #ENDIF
  #ENDAT
```

See Also:

#EMBED

#WHERE

#RESTRICT

##AT (insert code in an embed point)

#CREATE (create source file)

#CREATE(*file* **)**

#CREATE	Creates a disk file to receive generated source code.
file	A string constant, template symbol, or expression containing a DOS file specification. This may be a fully qualified DOS pathname.

The **#CREATE** statement creates a disk file to receive the source code generated by #GENERATE. If the *file* does not exist, it is created. If the *file* already exists, it is opened and emptied (truncated to zero length). If the *file* is already open, a source generation error is produced.

The *file* is automatically selected as the active source output destination.

Example:

```
#SET(%NewProgramFile,(%Application & '.$$$'))    #!Temp new program filename
#CREATE(%NewProgramFile)                         #!Create new program file
#GENERATE(%Program)                              #!Generate main program header
```

#CYCLE (cycle to top of loop)

#CYCLE

The **#CYCLE** statement immediately passes control back to the top of the #FOR or #LOOP structure in which it is enclosed to begin the next iteration. #CYCLE is only valid within a #FOR or #LOOP structure, else an error is generated during Template file pre-processing. #CYCLE acts as a #RETURN statement if issued from within a #GROUP inserted in the loop (unless it is within a #FOR or #LOOP structure completely contained within the #GROUP).

Example:

```
#SET(%StopFile,'CUSTOMER')
#FOR(%File)
  #IF (UPPER(%File) <> %StopFile)
    OPEN(%File)
    #CYCLE
  #ELSE
    #BREAK
  #ENDIF
#ENDFOR
```

#DEBUG (toggle debug generation)

#DEBUG(*value* **)**

#DEBUG	Toggles debug generation on and off.
value	A symbol containing zero (0) or one(1). When zero, debug generation is turned off. When one, debug generation is truned on.

The **#DEBUG** statement toggles debug generation on and off, and overrides the Debug Generation setting in the Application Options dialog. When this is checked, #DEBUG controls the specific sections to output to the text file, so the file can be restricted to a manageable size.

Example:

```
#SET(%DebugSet,1)
#DEBUG(%DebugSet)    #!Set debug generation on
IF Action = 1        #<!If adding a record
  SomeVariable = InitVariable
END
#SET(%DebugSet,0)
#DEBUG(%DebugSet)    #!Set debug generation off
```

#DECLARE (declare a user-defined symbol)

#DECLARE(*symbol* [, *parentsymbol*] [,*type*]) [, **MULTI**] [, **UNIQUE**] [, **SAVE**]

#DECLARE	Explicitly declares a user-defined symbol.
symbol	The name of the symbol being declared. This must meet all the requirements of a user-defined symbol. This must not be a #PROMPT symbol or a variable in the same scope.
parentsymbol	Specifies the parent of the *symbol*, indicating its value is dependent upon the current value in another symbol. This must be a multi-valued symbol. You may specify more than one *parentsymbol* if the *symbol* is dependent upon a set of symbols. This allows implicit multi-dimensional arrays.
type	The data type of the *symbol*: LONG, REAL, or STRING. If omitted, the data type is STRING.
MULTI	Specifies the *symbol* may contain multiple values.
UNIQUE	Specifies a multi-valued *symbol* that cannot contain duplicate values. The values are stored in ascending order. This implicitly declares the symbol as multi-valued, the MULTI attribute is not required.
SAVE	Specifies the value(s) in the *symbol* are saved between source generation sessions. A *symbol* with the SAVE attribute may only be declared in the #APPLICATION area.

The **#DECLARE** statement explicitly declares a user-defined *symbol*. This may contain a single value or multiple values. All user-defined symbols must be explicitly declared with #DECLARE except those declared on a #PROMPT statement and #GROUP parameters.

The MULTI attribute declares the *symbol* as multi-valued. This allows the #FIX, #FOR, #ADD, #DELETE, #SELECT, and #FREE statements to operate on the *symbol*.

The UNIQUE attribute ensures all instances of a multi-valued *symbol* to be unique and sorted in ascending sequence. When UNIQUE is specified, MULTI is not required. The #ADD statement builds the *symbol* values in sorted order and only allows a single instance of every value in the *symbol* when each entry is added.

If the #DECLARE statement contains one or more *parentsymbol* parameters, the user-defined *symbol* is dependent on the *parentsymbols*. This means a separate instance (or instances, if multi-valued) of the *symbol* is available for each instance of the *parentsymbol*. If there are no *parentsymbol* parameters, it is independent.

#DECLARE may be used to create dependent *symbols*. The *parentsymbol* must be a multi-valued symbol, whetherit is a built-in or user-defined symbol.

The SAVE attribute causes a *symbol's* value(s) to be saved at the end of source generation and restored when the #DECLARE statement is eXEcuted at the beginning of the next source generation session. A *symbol* with the SAVE attribute may only be declared in the #APPLICATION section.

Example:

```
#APPLICATION('Sample One')
#DECLARE(%UserSymbol),SAVE                   #!Value saved after generation
                                             #! and restored for next generation
#DECLARE(%ModuleFile,%Module),UNIQUE,MULTI  #!Level-1 dependent symbol
#DECLARE(%ModuleFilePut,%ModuleFile)        #!Level-2 dependent symbol
#DECLARE(%ModuleFileDelete,%ModuleFile)     #!Second Level-2 dependent symbol
```

See Also:

#FIX

#FOR

#ADD

#DELETE

#FREE

#DEFAULT (default procedure starting point)

#DEFAULT

procedure

#ENDDEFAULT

#DEFAULT	Begins a default procedure declaration section.
procedure	Default procedure in .TXA file format.
#ENDDEFAULT	Terminates the default procedure declaration.

The **#DEFAULT** structure contains a single default *procedure* declaration in .TXA format as generated by the Application Generator's Export function. #DEFAULT may only be placed at the end of a #PROCEDURE section. You may have multiple #DEFAULT structures for a single #PROCEDURE. The enclosed *procedure* section of a .TXA file should contain a procedure of the preceding #PROCEDURE's type. The recommended way to create a #DEFAULT structure is to edit the default procedure in the template registry, and then export the template as text, which creates a .TXA file.

Example:

```
#DEFAULT
NAME DefaultForm
[COMMON]
DESCRIPTION 'Default record update'
FROM Clarion Form
[PROMPTS]
%WindowOperationMode STRING  ('Use WINDOW setting')
%INISaveWindow LONG  (1)
[ADDITION]
NAME Clarion SaveButton
[INSTANCE]
INSTANCE 1
PROCPROP
[PROMPTS]
%InsertAllowed LONG  (1)
%InsertMessage @S30  ('Record will be Added')
%ChangeAllowed LONG  (1)
%ChangeMessage @S30  ('Record will be Changed')
%DeleteAllowed LONG  (1)
%DeleteMessage @S30  ('Record will be Deleted')
%MessageHeader LONG  (0)
[ADDITION]
NAME Clarion CancelButton
[INSTANCE]
INSTANCE 2
```

```
[WINDOW]
FormWindow WINDOW('Update Records...'),AT(18,5,289,159),CENTER,SYSTEM,GRAY,MDI
       BUTTON('OK'),AT(5,140,40,12),USE(?OK),DEFAULT,#SEQ(1),#ORIG(?OK),#LINK(?Cancel)
       BUTTON('Cancel'),AT(50,140,40,12),USE(?Cancel),#SEQ(2),#ORIG(?Cancel)
       STRING(@S40),AT(95,140,,),USE(ActionMessage)
           END
#ENDDEFAULT
```

#DEFINE (user defined macro)

#DEFINE (*%macro name* **)**

statements

#ENDDEFINE

#DEFINE	Defines a user defined macro that can be expanded using #EXPAND.
%macro name	A user defined symbol used as the macro's identifier.
statements	A sequence of statements.
#ENDDEFINE	Terminates the definition of a user defined macro.

#DEFINE defines the beginning of a section of code which can be expanded into generated code using the #EXPAND statement. A macro may not be defined with any parameters. #DEFINE must be terminated with #ENDDEFINE. Using #DEFINE and #EXPAND to define and expand macros for commonly used code sections will lead to cleaner template code while hiding unneeded target code from immediate view.

Example:

```
#DEFINE(%InRangeCheck)
  IF %ControlUse <= TODAY()
   SELECT(?%ControlUse)
  END
#ENDDEFINE
```

See Also: #EXPAND

#DELETE (delete a multi-valued symbol instance)

#DELETE(*symbol [*, *position*] **)**

#DELETE	Deletes the value from one instance of a multi-valued user-defined symbol.
symbol	A multi-valued user-defined symbol.
position	An integer constant or symbol containing the instance number in the *symbol*. Instance numbering begins with one (1). If omitted, the default is the current fiXEd instance.

The **#DELETE** statement deletes the value from one instance of a multi-valued user-defined symbol. If there are any symbols dependent upon the *symbol*, they are also cleared. If this is the last instance in the *symbol*, the instance is removed. You can get the current instance number to which a symbol is fiXEd by using the **INSTANCE(%symbol)** built-in template procedure.

Example:

```
#DECLARE(%ProcFilesPrefix),MULTI    #!Declare multi-valued symbol
#ADD(%ProcFilesPrefix,'SAV')        #!Add a value
#ADD(%ProcFilesPrefix,'BAK')        #!Add a value
#ADD(%ProcFilesPrefix,'PRE')        #!Add a value
#ADD(%ProcFilesPrefix,'QUE')        #!Add a value
#!%ProcFilesPrefix contains: SAV, BAK, PRE, QUE
#DELETE(%ProcFilesPrefix,1)         #!Delete first value (SAV)
#!%ProcFilesPrefix contains: BAK, PRE, QUE
#FIX(%ProcFilesPrefix,'PRE')        #!Fix to a value
#DELETE(%ProcFilesPrefix)           #!Delete it
#!%ProcFilesPrefix contains: BAK, QUE
```

See Also:	#DECLARE
	#ADD

#DELETEALL (delete multiple multi-valued symbol instances)

#DELETEALL(*symbol, expression* **)**

#DELETEALL	Deletes the values from specified instances of a multi-valued user-defined symbol.
symbol	A multi-valued user-defined symbol.
expression	An expression that defines the instances to delete.

The **#DELETEALL** statement deletes all values from the *symbol* that meet the *expression*.

Example:

```
#DECLARE(%ProcFilesPrefix),MULTI      #!Declare multi-valued symbol
#ADD(%ProcFilesPrefix,'SAV')          #!Add a value
#ADD(%ProcFilesPrefix,'BAK')          #!Add a value
#ADD(%ProcFilesPrefix,'PRE')          #!Add a value
#ADD(%ProcFilesPrefix,'BAK')          #!Add a value
#ADD(%ProcFilesPrefix,'QUE')          #!Add a value
#!%ProcFilesPrefix contains: SAV, BAK, PRE, BAK, QUE
#DELETEALL(%ProcFilesPrefix,'BAK')    #!Delete all BAK instances
#!%ProcFilesPrefix now contains: SAV, PRE, QUE
```

See Also: #DECLARE

#ADD

#DISPLAY (display-only prompt)

#DISPLAY([*string*] **)** [, **AT( )**] [, **PROP(** *name,value* **)**]

#DISPLAY	Displays a string constant on a properties window.
string	A string expression containing the text to display.
AT	Specifies the size and position of the *string* display area in the window, allowing multiple lines of text. This attribute takes the same parameters as the Clarion language AT attribute. If a width and height are specified in the AT attribute, then the *string* will span multiple lines.
PROP	Specifies a property to assign to the prompt text. *Name* designates the property name equate (Example: PROP:FontColor) and *value* is the value assigned to the named property (Example: 0FFFFFFH)

The **#DISPLAY** statement displays the *string* on a properties window. If the *string* is omitted, a blank line is displayed. The display updates whenever the value in the *string* changes. #DISPLAY is not valid in a #MODULE section.

Example:

```
#DISPLAY()                                      #!Display a blank line
#DISPLAY('Ask programmer to input some')      #!Display a string
#DISPLAY(' ABC Version: '&%ABCVersion), AT(10,,170), PROP(PROP:FontColor, 0FFFFFFH)
#PROMPT(' specific value',@s20),%InputSymbol
```

See Also: #PROMPT

#GROUP

#BOXED

#ENABLE

#BUTTON

#EMBED (define embedded source point)

#EMBED(*identifier* [*, descriptor*] **)** [, *symbol*] [, **HLP(** *helpid* **)**] [, **DATA**] [, **HIDE**] [, **DESCRIPTION**(*text*)]

[, **WHERE(** *expression* **)**] [, **MAP(** *mapsymbol, description* **)**] [, **LABEL**] [, **NOINDENT**]
[, **DEPRECATED**] [, **LEGACY**] [, **PREPARE(** *parameters* **)**] [, **TREE(** *displaytext)*]

#EMBED	Identifies an explicit position in the Template where the programmer may place their own source code.
identifier	A user-defined template symbol which identifies the embedded source code point for the Application Generator.
descriptor	A string constant containing a description of the embedded source code's position in the Template. This is the string displayed in the list of available embedded source code windows for a procedure Template (normally omitted if the HIDE attribute is present).
symbol	A multi-valued template symbol or a literal string. You may have multiple *symbols* on a single #EMBED statement. This may also have a description to specify the text to display in the embedded source tree appended to it with square brackets (i.e. %symbol[%description]).
HLP	Specifies on-line help is available for the #EMBED.
helpid	A string constant containing the identifier to access the Help system. This may be either a Help keyword or "context string."
DATA	Specifies the embed point is in a data section, so the Text Editor's Window and Report Formatters can be used.
HIDE	Specifies the source code point does not appear in the tree of available embedded source code points. Therefore, the #EMBED is only available for #CODE, #CONTROL, or #EXTENSION code generation.
DESCRIPTION	Specifies *text* that will be displayed at the embed tree node specified by the *location*.
WHERE	Specifies the #EMBED is available only for those instances of the *symbol* where the *expression* is true.
expression	An expression that specifies the condition.
MAP	Maps the *description* to the *symbol* for display in the embedded source tree. You may have as many MAP attributes as there are *symbols*.
mapsymbol	Names which of the #EMBED *symbols* the MAP references.
description	An expression that specifies the text to display in the embedded source tree.

LABEL	Specifies the embed point is generated in column one (1) so the Text Editor will correctly color-code the labels.
NOINDENT	Specifies the code generated into the embed point retains the indention from its source (either generated or entered in by the programmer) source in column 1 is generated into column 1, no matter where the #EMBED is placed.
DEPRECATED	Specifies the #EMBED is displayed in the list of available embed points only if there is already code in it. This indicates an embed point which has been superceded and is only present to allow the existing code in it to be moved to a new embed or deleted.
LEGACY	Specifies to show the embed only when you press the legacy button on the embed tree.
PREPARE	Executes preparatory code before evaluating the #EMBED *symbol s*. The n_{th} *parameter* to PREPARE is evaluated immediately before the n_{th} *symbol* is evaluated. *parameters* A comma-delimited list of expressions or calls to #GROUPs. You may have as many PREPARE *parameters* as there are *symbols* on the #EMBED.
TREE	Specifies what will be added to the embeds dialog tree. This overrides the MAP attribute. *displaytext* A comma-delimited list of strings (constraints or variables) containing tree level text, color, and priority information. Each tree level may be a separate entry or contained in a single separated by vertical bar (|) character. Following the text to display you may have optional color and priority information contained in curly braces, in the following format: **'TextToDisplay{{Color(0FFh),priority(5000)}'** The color information may be either a number or the text for a color EQUATE from the EQUATES.CLW file.

#EMBED identifies an explicit position in the Template where the programmer may call a procedure, generate code from a code template, or place their own custom embedded source code within the procedure. The Application Generator prompts the programmer for the procedure to call, or the code template to use, or calls the Text Editor to allow the programmer to write the embedded source code. #EMBED is also used as the destination of all the source automatically generated by #CODE, #CONTROL, and #EXTENSION template sections. If no code is written in the embedded source code point by the programmer or any code template, control template, or extension template, no code is generated.

In a #PROCEDURE section, the source code is automatically placed in the exact column position at which #EMBED is located within the Template. If #EMBED is directly placed in the data section of a #PROGRAM, #MODULE, or #PROCEDURE, it must be in column one (1) of the Template file (so the embedded code may contain data labels). If the #EMBED statement has the DATA atribute, the Window and Report Formatters in the Text Editor are available for use. In executable code sections, #EMBED may be placed in column one, but that is not required.

#EMBED is valid in a #GROUP section, however, this should be used with care. Since it is possible for a #GROUP to be recursive (call itself), it is possible to create embedded source code that is repeated within each iteration of the recursive #GROUP's generated code. The source code is generated in the same relative column position as the code generated from the #GROUP. A #EMBED using the *symbol* attribute is used within a #FOR statement to allow a different piece of embedded source to be inserted for each instance of the *symbol*. It can also be used within #FOR, #LOOP, and/or recursive #GROUPs for the current instance of the symbol (if it has been #FIXed). The MAP attribute allows you to replace a *description* for the *symbol* in the embedded source tree.

Example:

```
#PROCEDURE(SampleProc,'This is a sample procedure'),WINDOW
#EMBED(%DataSection,'Data Section Source Code Window'),DATA #!Source code in column 1
  CODE                !Begin executable code
  #EMBED(%SetupProc,'Code Section Source Code Window 1')    #!Source code in column 3
  OPEN(Screen)        !Open window
  ACCEPT              !Event handler
    CASE SELECTED()  !Handle field-selection events
    #FOR(%Control)
    OF %Control
      #EMBED(%ScreenFieldSetupEmbed,'Field Selected Embed'),%Control
    #ENDFOR
    END
    CASE ACCEPTED()  !Handle field-action events
    #FOR(%Control)
    OF %Control
      #EMBED(%ScreenFieldEditEmbed,'Field Accepted Embed'),%Control
    #ENDFOR
    END
  END
#EMBED(%CustomRoutines,'Code Section Source Code Window 2'),LABEL  #!Source in col 1
```

#EMPTYEMBED (generate empty embed point comments)

#EMPTYEMBED(*text* [, *condition*] **)**

#EMPTYEMBED	Generates comments into empty embed points.
text	A string constant or constant expression containing the text to place in the empty embed point.
condition	An expression that, when true, allows the comments to generate.

The **#EMPTYEMBED** statement specifies that comments generate into all embed points in which the user has not entered code. This will not generate comments for embed points in which the user has entered code or in which the templates have generated code.

The output *condition* is usually the value of a global prompt.

The comment *text* may use the %EmbedID, %EmbedDescription, and %EmbedParameters built-in symbols to identify the embed point:

%EmbedID	The current embed point's identifying symbol.
%EmbedDescription	The current embed point's description.
%EmbedParameters	The current embed point's current instance, as a comma-delimited list.

Example:

```
#! This example is complete extension template, ready to use
#EXTENSION(EmptyEmbeds,'Empty Embed Comments'),APPLICATION
#PROMPT('Generate Empty EMBED Comments',CHECK),%EmptyEmbeds
#EMPTYEMBED('!Embed: ' & %EmbedDescription & ' ' & %EmbedParameters,%EmptyEmbeds)
```

See Also: #PREEMBED

#POSTEMBED

#ENABLE (enable/disable prompts)

#ENABLE(*expression* **)** [, **CLEAR**] [, **SECTION**]

prompts

#ENDENABLE

#ENABLE	Begins a group of *prompts* which may be enabled or disabled based upon the evaluation of the *expression*.
expression	The expression which controls the prompt enable/disable.
CLEAR	Specifies the *prompts* symbol values are cleared when disabled.
SECTION	Specifies all AT() attributes for the *prompts* are positioned relative to the start of the #ENABLE section.
prompts	One or more #PROMPT, #BUTTON, #DISPLAY, #ENABLE, and/or #VALIDATE statements.
#ENDENABLE	Terminates the group of *prompts*.

The **#ENABLE** structure contains *prompts* which may be enabled or disabled based upon the evaluation of the *expression*. If the *expression* is true, the *prompts* are enabled, otherwise they are disabled. The *prompts* appear dimmed when disabled and the programmer may not enter data in them.

Example:

```
#PROMPT('Pick One',OPTION),%InputChoice          #!Mutually exclusive options
#PROMPT('Choice One',RADIO)
#PROMPT('Choice Two',RADIO)
#ENABLE(%InputChoice = 'Choice Two')
  #PROMPT('Screen Field',CONTROL),%SomeField     #!Enabled only for Choice Two
  #VALIDATE(%ScreenFieldType = 'LIST','Must select a list box')
#ENDENABLE
```

See Also:	#PROMPT #GROUP #BOXED #BUTTON

#EQUATE (declare and assign value to a user-defined symbol)

#EQUATE(*symbol*,*value* **)**

#EQUATE	Declares and assigns a value to a single-valued user-defined symbol.
symbol	A single-valued user-defined symbol. This must not have been previously declared.
value	A built-in or user-defined symbol, string constant, or an expression.

The **#EQUATE** statement declares the *symbol* and assigns the *value* to the *symbol*. This is directly equivalent to a #DECLARE statement followed by a #SET to assign it a value.

If the *value* parameter contains an expression, you may perform mathematics during source code generation. The expression may use any of the arithmetic, Boolean, and logical operators documented in the *Language Reference*. If the modulus division operator (%) is used in the expression, it must be followed by at least one blank space (to explicitly differentiate it from the Template symbols). Logical expressions always evaluate to 1 (True) or 0 (False). Clarion language procedure calls (those supported in EVALUATE()) and built-in template procedures are allowed.

Example:

```
#EQUATE(%NetworkApp,'Network')
#EQUATE(%MySymbol,%Primary)
```

See Also: #DECLARE

#SET

#ERROR (display source generation error)

#ERROR(*message* **)**

#ERROR	Displays a source generation error.
message	A string constant, user-defined symbol, or expression containing an error message to display in the Source Generation window.

#ERROR displays a *message* in the Source Generation window. This could be information for the user. It may also alert the user that they made some error which will cause the procedure Template to generate invalid source code which could create compiler errors.

When a #ERROR statement is encountered at source code generation time, its message is displayed. The user may choose to abort the compile and link process, or continue on to the compiler.

Example:

```
#PROCEDURE(SampleProc,'This is a sample procedure')
#PROMPT('Access Key',KEY),%SampleAccessKey
  #IF(%SampleAccessKey = %NULL)           #!IF the user did not enter a Key
    #SET(%ErrorSymbol,(%Procedure & ' Access Key blank')
    #ERROR(%ErrorSymbol)
    #ERROR('This error is Fatal -- DO NOT CONTINUE')
   #ABORT
  #ENDIF
```

#EXPAND (expand a user defined macro)

#EXPAND (*%macro name* **)**

#EXPAND	Expands a user defined macro.
%macro name	A previously declared macro symbol.

#EXPAND is used to expand a previously defined macro into the generated source code. Using #DEFINE and #EXPAND to define and expand macros for commonly used code sections will lead to cleaner template code while hiding unneeded target code from immediate view.

Example:

```
#EXPAND(%InRangeCheck)

The above template code will cause the following code to be generated:

  IF %ControlUse <= TODAY()
   SELECT(?%ControlUse)
  END
```

See Also: #DEFINE

#EXPORT (export symbol to text)

#EXPORT([*symbol*] **)**

#EXPORT	Creates a .TXA text file from a *symbol*.
symbol	The template symbol to export (%Module, %Procedure, or %Program). If omitted, the current application is exported.

#EXPORT outputs .TXA script text for the *symbol* to the current output file (see #CREATE or #OPEN). This .TXA file may then be used for importing to other Clarion applications.

Example:

```
#OPEN('MyExp.TXA')
#FOR(%Procedure)
  #EXPORT(%Procedure)
#ENDFOR
```

See Also:	#CREATE
	#OPEN
	#IMPORT

#EXTENSION (define an extension template)

```
#EXTENSION( name, description [, target ] ) [, MULTI ] [, DESCRIPTION( expression ) ] ]
[, SHOW ] [, PRIMARY( message [, flag ] ) ] [,           | APPLICATION( [ child(chain) ] )    | ]
                                                         | PROCEDURE                          |
          [, REQ( addition [,| BEFORE         | ] ) [,   | FIRST  | ] [, PRIORITY( number )]
                             | AFTER|                    | LAST   |
```

#EXTENSION	Begins an extension template that generates code into embedded source code points to add some functionality not associated with specific controls.
name	The label of the extension template. This must be a valid Clarion label.
description	A string constant describing the extension template.
target	A string constant that specifies the source language the extension template generates. If omitted, it defaults to Clarion.
MULTI	Specifies the #EXTENSION may be used multiple times in a given application or procedure.
DESCRIPTION	Specifies the display description of a #EXTENSION that may be used multiple times in a given application or procedure.
expression	A string constant or expression that contains the description to display.
SHOW	Specifies the #EXTENSION prompts are placed on the procedure properties window.
PRIMARY	Specifies a primary file for the extension must be placed in the procedure's File Schematic.
message	A string constant containing a message that appears in the File Schematic next to the #EXTENSION's Primary file.
flag	Either OPTIONAL (the file is not required), OPTKEY (the key is not required), or NOKEY (the file is not required to have a key).
APPLICATION	Tells the Application Generator to make the #EXTENSION available only at the global level.
child(chain)	The name of a #EXTENSION with the PROCEDURE attribute to automatically populate into every generated procedure when the #EXTENSION with the APPLICATION attribute is populated.
PROCEDURE	Tells the Application Generator to make the #EXTENSION available only at the local level.
REQ	Specifies the #EXTENSION requires a previously placed #CODE, #CONTROL, or #EXTENSION before it may be used.
addition	The name of the previously placed #CODE, #CONTROL, or #EXTENSION.
BEFORE	Legacy attribute, replaced by PRIORITY.

AFTER	Legacy attribute, replaced by PRIORITY.
FIRST	Equivalent to PRIORITY(1).
LAST	Equivalent to PRIORITY(10000).
PRIORITY	Specifies the order inwhich the #EXTENSION is generated. The lowest value generates first.
number	An integer constant in the range 1 to 10000.

#EXTENSION defines the beginning of an extension template containing code to generate into the application or procedure to provide some functionality not directly associated with any control. A #EXTENSION section may contain Template and/or target language code. The #EXTENSION section is terminated by the first occurrence of a Template Code Section (#PROGRAM, #MODULE, #PROCEDURE, #CONTROL, #CODE, #EXTENSION, or #GROUP) statement, or the end of the file. Within a single Template set, separate #EXTENSION sections may not be defined with the same *name*.

#EXTENSION can only generate code into #EMBED embedded source code points using the #AT/#ENDAT structure. A #EXTENSION section may contain #PROMPT statements to prompt for the values needed to generate proper source code. These prompts appear when you edit an Extension from the Extensions button in the environment. It may also contain #EMBED statements which become active only if the #EXTENSION section is used.

#RESTRICT can restrict appearance of the #EXTENSION in the list of available extensions based on an expression or Template language statements.

Example:

```
#EXTENSION(Security,'Add password'),PROCEDURE
  #PROMPT('Password File',FILE),%PasswordFile,REQ
  #PROMPT('Password Key',KEY(%PasswordFile)),%PasswordFileKey,REQ
  #PROMPT('Password Field',COMPONENT(%PasswordFileKey)),%PasswordFileKeyField,REQ
  #AT(%DataSectionBeforeWindow)
LocalPswd   STRING(10)
SecurityWin WINDOW
             ENTRY(@s10),USE(LocalPswd),REQ,PASSWORD
             BUTTON('Cancel'),KEY(EscKey),USE(?CancelPswd)
            END
  #ENDAT
  #AT(%ProcedureSetup)
OPEN(SecurityWin)
ACCEPT
  CASE ACCEPTED()
  OF ?LocalPswd
    %PasswordFileKeyField = LocalPswd
    GET(%PasswordFile,%PasswordFileKey)
    IF NOT ERRORCODE()
      LocalPswd = 'OK'
    END
  BREAK
  OF ?CancelPswd
   CLEAR(LocalPswd)
   BREAK
  END
END
CLOSE(SecurityWin)
IF LocalPswd <> 'OK' THEN RETURN.
  #ENDAT
```

See Also:

#EMBED

#WHERE

#RESTRICT

#AT (insert code in an embed point)

#FIELD (control prompts)

#FIELD, **WHERE(** *expression* **)**

prompts

#ENDFIELD

#FIELD	Begins a control prompts section.
WHERE	Specifies the #FIELD is used only for those instances where the *expression* is true.
expression	An expression that specifies the condition for use.
prompts	Prompt (#PROMPT, #BUTTON, etc.) statements.
#ENDFIELD	Terminates the section.

The **#FIELD** structure contains *prompts* for controls that are populated onto a window. These *prompts* appear in the Actions... dialog.

The list of field prompts appearing in the Actions... dialog is built in the following manner:

1. #CONTROL prompts.
2. #PROCEDURE-level #FIELD prompts (also from inserted #GROUPs).
3. #PROCEDURE-level #FIELD prompts from active #EXTENSION sections.
4. #CONTROL-level #FIELD prompts.
5. #CODE-level #FIELD prompts.

The values the user inputs into the #FIELD prompts are used to generate the source to govern the behavior of the control.

Example:

```
#FIELD, WHERE(%ControlType = 'BUTTON')
  #PROMPT('Enter procedure call',PROCEDURE),%ButtonProc
#ENDFIELD
```

See Also: #PROMPT

#VALIDATE

#ENABLE

#DISPLAY

#BUTTON

#FIND ("super-fix" multi-value symbols)

#FIND(*symbol*, *fixsymbol* [, *limit*] **)**

#FIND	FIXes all multi-valued parent symbols to values that point to a single child instance.
symbol	A multi-valued symbol.
fixsymbol	A symbol or expression containing the value to fix the *symbol* to.
limit	A parent symbol which limits the search scope to the children of the *limit* symbol.

The **#FIND** statement finds the first instance of the *fixsymbol* contained within the *symbol* then fiXEs it and all the "parent" symbols on which the *symbol* is dependent to the values that "point to" the value of the *fixsymbol* contained in the *symbol*. This is done so that all the symbol dependencies are aligned and you can reference other symbols dependent on "parent" symbols of the *symbol*.

For example, assume %ControlUse contains CUS:Name. The #FIND(%Field,%ControlUse) statement:

- Finds the first instance of %Field that matches the current value in %ControlUse (the first instance of CUS:Name in %Field) in the current procedure.
- FIXes %Field to that value (CUS:Name).
- FIXes %File to the name of the file containing that field (Customer).
- This allows the Template code to reference other the symbols dependent upon %File (like %FilePre to get the file's prefix).

The *fixsymbol* must contain a valid instance of one of the *symbol's* multiple values. If the *fixsymbol* does not contain a valid instance, the *symbol* is cleared and contains no value when referenced.

Example:

```
#FIND(%Field,%ControlUse)      #!FIXEs %Field and %File to %ControlUse parents
```

See Also: #SELECT

#FIX

#FIX (fix a multi-value symbol)

#FIX(*symbol*, *fixsymbol* **)**

#FIX	FIXes a multi-valued symbol to the value of a single instance.
symbol	A multi-valued symbol.
fixsymbol	A symbol or expression containing the value to fix the *symbol* to.

The **#FIX** statement fixes the current value of the multi-valued *symbol* to the value contained in the *fixsymbol*. This is done so that one instance of the *symbol* may be referenced outside a #FOR loop structure, or so you can reference the symbols dependent upon the multi-valued *symbol*.

The *fixsymbol* must contain a valid instance of one of the *symbol's* multiple values. If the *fixsymbol* does not contain a valid instance, the *symbol* is cleared and contains no value when referenced. Unless #ADD has been used to add a new value and fix to that instance, #FIX or #SELECT must be used to set the value in a *symbol* before it contains any value for Template processing outside of a #FOR loop.

#FIX is completely independent of #FOR in that #FOR always loops through every instance of the *symbol*, whether there is a previous #FIX for that *symbol* or not. If there is a previous #FIX statement for that *symbol* before the #FOR loop, that *symbol* reverts to that previous *fixvalue* after the #FOR terminates.

If #FIX is used within a #FOR structure, the scope of the #FIX is limited to within the #FOR in which it is used. It does not change the #FOR symbol's iteration value if both the #FOR and #FIX happen to use the same symbol.

Example:

```
#SET(%OneFile,'HEADER')         #! Put values into two User-defined symbols
#SET(%TwoFile,'DETAIL')
#FIX(%File,%OneFile)            #! %File refers to 'HEADER'
#FOR(%File)                     #! %File iteratively refers to all file names
  #FIX(%File,%TwoFile)          #! %File refers to 'DETAIL'
#ENDFOR
                                #! %File refers to 'HEADER' again
```

See Also: #SELECT

#FOR (generate code multiple times)

#FOR(*symbol* **)** [, **WHERE(** *expression* **)**] [, **REVERSE**]

statements

#ENDFOR

#FOR	Loops through all instances of a multi-valued symbol.
symbol	A multi-valued symbol.
WHERE	Specifies the *statements* in the #FOR loop are executed only for those instances of the *symbol* where the *expression* is true.
expression	An expression that specifies the condition for execution.
REVERSE	Specifies the #FOR loops through the instances of the *symbol* in reverse order.
statements	Target and/or Template Language statements.
#ENDFOR	Terminates the #FOR structure.

#FOR is a loop structure which generates its *statements* once for each value contained in its *symbol* during source code generation. If there are no values in the *symbol*, no code is generated. #FOR must be terminated by **#ENDFOR**. If there is no #ENDFOR, an error message is issued during Template file pre-processing. A #FOR loop may be nested within another #FOR loop.

The #FOR loop begins with the first instance of the *symbol* (or last, if the **REVERSE** attribute is present) and processes through all instances of the *symbol*--it is not affected by any #FIX statements. If the **WHERE** attribute is present, the #FOR *statements* are executed only for those instances of the *symbol* where the *expression* is true. This creates a conditional #FOR loop.

Since #FOR is a loop structure, the #BREAK and #CYCLE statements may be used to control the loop. #BREAK immediately terminates #FOR loop processing and continues with the statement following the #ENDFOR that terminates the #FOR. #CYCLE immediately returns control to the #FOR statement to continue with the next instance of the *symbol*.

Example:

```
#FOR(%ScreenField),WHERE(%ScreenFieldType = 'LIST')
  #INSERT(%ListQueueBuild)            #!Generate only for LIST controls
#ENDFOR
```

See Also: #BREAK

#CYCLE

#FREE (free a multi-valued symbol)

#FREE(*symbol* **)**

#FREE	Clears all instances of a multi-valued user-defined symbol.
symbol	A multi-valued user-defined symbol.

The **#FREE** statement clears all instances of a multi-valued user-defined symbol. If there are any symbols dependent upon the *symbol*, they are also cleared.

Example:

```
#DECLARE(%ProcFilesPrefix),MULTI      #!Declare multi-valued symbol
#ADD(%ProcFilesPrefix,'SAV')          #!Add a value
#ADD(%ProcFilesPrefix,'BAK')          #!Add a value
#ADD(%ProcFilesPrefix,'PRE')          #!Add a value
#ADD(%ProcFilesPrefix,'BAK')          #!Add a value
#ADD(%ProcFilesPrefix,'QUE')          #!Add a value
                #!%ProcFilesPrefix contains: SAV, BAK, PRE, BAK, QUE
#DELETEALL(%ProcFilesPrefix,'BAK')    #!Delete all BAK instances
                #!%ProcFilesPrefix now contains: SAV, PRE, QUE
#FREE(%ProcFilesPrefix)               #!Free the symbol
                #!%ProcFilesPrefix now contains nothing
```

See Also:	#DECLARE #ADD

#GENERATE (generate source code section)

#GENERATE(*section* **)**

#GENERATE	Generates a section of the application.
section	One of the following built-in symbols: %Program, %Module, or %Procedure. This symbol indicates the portion of the application to generate.

The **#GENERATE** statement generates the source code for the specified *section* of the application by executing the Template Language statements contained within that *section*. #GENERATE should only be used within the #APPLICATION or a #UTILITY section of the Template.

When *section* is:

%Program	The #PROGRAM section of the Template is generated.
%Module	The appropriate #MODULE section of the Template is generated.
%Procedure	The appropriate #PROCEDURE section of the Template for the current value of %Procedure is generated.

Example:

```
#GENERATE(%Program)                               #!Generate program header
#FOR(%Module)                                     #!
  #GENERATE(%Module)                              #!Generate module header
  #FOR(%ModuleProcedure)                          #!For all procs in module
    #FIX(%Procedure,%ModuleProcedure)             #!Fix current procedure
    #GENERATE(%Procedure)                         #!Generate procedure code
  #ENDFOR                                         #!EndFor all procs in module
#ENDFOR                                           #!EndFor all modules
```

#GLOBALDATA (default global data declarations)

#GLOBALDATA

declarations

#ENDGLOBALDATA

#GLOBALDATA	Begins a default global data declaration section.
declarations	Data declarations.
#ENDGLOBALDATA	Terminates the default global data declarations.

The **#GLOBALDATA** structure contains default data *declarations* global to the program. The *declarations* are then available to the programmer through the *Data* button on the **Global Properties** dialog. #GLOBALDATA may be placed in a #PROGRAM, #PROCEDURE, #CODE, #CONTROL, or #EXTENSION section of the Template. The *declarations* will appear in the global data section of the generated source code.

Example:

```
#GLOBALDATA
Action   BYTE         !Disk action variable
TempFile CSTRING(65)  !Temporary filename variable
#ENDGLOBALDATA
```

#GROUP (reusable statement group)

#GROUP(*symbol* [, [*type*] *parameters* [= *default*]] **)** [, **AUTO**] [, **PRESERVE**] [, **HLP(** *helpid* **)**]

#GROUP	Begins a section of template code that may be inserted into another portion of the template.
symbol	A user-defined symbol used as the #GROUP's identifier.
type	The data type of a passed *parameter*: LONG, REAL, STRING, or * (asterisk). An asterisk (*) indicates it is a variable-parameter (passed by address), whose value may be changed by the #GROUP. LONG, REAL, and STRING indicates it is a value-parameter (passed by value), whose value is not changed by the #GROUP. If *type* is omitted, the *parameter* is a passed as a STRING.
parameters	User-defined symbols by which values passed to the #GROUP are referenced. You may pass multiple *parameters*, each separated by commas, to a #GROUP. All specified *parameters* must be passed to the #GROUP; they may be omitted only if a *default* is supplied.
=default	The value passed for an omitted *parameter*.
AUTO	Opens a new scope for the group. This means that any #DECLARE statements in the #GROUP would not be available to the #PROCEDURE being generated. Passing *parameters* to a #GROUP implicitly opens a new scope.
PRESERVE	Preserves the current fixed instances of all built-in multi-valued symbols when the #GROUP is called and restores all those instances when the #GROUP code terminates.
HLP	Specifies on-line help is available.
helpid	A string constant containing the identifier to access the Help system. This may be either a Help keyword or "context string."

#GROUP defines the beginning of a section of code which is generated into the source. A #GROUP section may contain Template and/or target language code. The #GROUP section is terminated by the first occurrence of a Template Code Section (#PROGRAM, #MODULE, #PROCEDURE, #CONTROL, #CODE, #EXTENSION, or #GROUP) statement, or the end of the file. Within a single Template, separate #GROUP sections may not be defined with the same *symbol*. The *parameters* passed to a #GROUP fall into two categories: **value-parameters** and **variable-parameters**.

Value-parameters are declared as user-defined symbols, with an optional *type* and are "passed by value" (a copy of the value is passed) Either symbols or expressions may be passed as value-parameters. When a multi-valued symbol is passed as a value-parameter, only the current instance is passed.

Variable-parameters are declared as user-defined symbols with a prepended asterisk (*****) (and no *type*). A variable-parameter is "passed by address" and any change to its value by the #GROUP code changes the value of the passed symbol. Only symbols may be passed to a #GROUP as variable-parameters. When a multi-valued symbol is passed as a variable-parameter, all instances are passed.

The statements contained in the #GROUP section are generated by the #INSERT or #CALL statements. A #GROUP may contain #EMBED statements to define embedded source code points. A #GROUP may contain #PROMPT statements to obtain programmer input.

A #GROUP may act as a function if the #RETURN statement which passes control back from the #GROUP has a parameter. The value is returned to the CALL built-in procedure or directly to the expression in which the #GROUP is called as a function. If the #GROUP is called without the CALL built-in procedure and takes no parameters, open and close parentheses must be appended to the #GROUP symbol. For example, you may either place CALL(%MyGroup) in an expression or just %MyGroup().

Example:

```
#GROUP(%GenerateFormulas)           #!A #GROUP without parameters
  #FOR(%Formula)
    #IF(%FormulaComputation)
%Formula = %FormulaComputation
    #ELSE
IF(%FormulaCondition)
  %Formula = %FormulaTrue
ELSE
  %Formula = %FormulaFalse
END
    #ENDIF
  #ENDFOR

#GROUP(%ChangeProperty,%MyField,%Property,%Value) #!A #GROUP that receives parameters
%MyField{%Property} = '%Value' #<!Change the %Property of %MyField

#!A #GROUP that receives a variable-parameter and a value-parameter:
#GROUP(%SomeGroup, * %VarParm, LONG %ValParm)
```

See Also:

#INSERT

#RETURN

CALL

#HELP (specify template help file)

#HELP(*helpfile* **)**

#HELP	Specifies the Template's help file.
helpfile	A string constant containing the name of the template's help file.

The **#HELP** statement specifies a *helpfile* which is used by this template. Once specified, the *helpfile* is used to access the help topics specified by the help id's in all HLP attributes in the template.

Example:

```
#HELP('Template.HLP')
```

#IF (conditionally generate code)

#IF(*expression* **)**

statements

[**#ELSIF(** *expression* **)**

statements]

[**#ELSE**

statements]

#ENDIF

#IF	A conditional execution structure.
expression	Any Template Language expression which can evaluate to false (blank or zero) or true (any other value). The expression may contain Template symbols, constant values, and any of the arithmetic, Boolean, and logical operators documented in the *Language Reference*. Procedure calls are allowed. If the modulus division operator (%) is used in the expression, it must be delimited by at least one blank space on each side (to explicitly differentiate it from the Template symbols).
statements	One or more Clarion and/or Template Language statements.
#ELSIF	Provides an alternate *expression* to evaluate when preceding #IF and #ELSIF *expressions* are false.
#ELSE	Provides alternate *statements* to execute when all preceding #IF and #ELSIF *expressions* are false.
#ENDIF	Terminates the #IF structure.

#IF selectively generates a group of *statements* depending on the evaluation of the *expression(s)*. The #IF structure consists of a #IF statement and all statements following it until the structure is terminated by **#ENDIF.** If there is no #ENDIF, an error message is issued during Template file pre-processing. #IF structures may be nested within other #IF structures.

#ELSIF and **#ELSE** are logical separators which separate the #IF structure into *statements* groups which are conditionally generated depending upon the evaluation of the *expression(s)*. There may be multiple #ELSIF statements within one #IF structure, but only one #ELSE.

When #IF is encountered during code generation:

- If the *expression* evaluates as true, only the *statements* following #IF are generated, up to the next following #ELSIF, #ELSE, or #ENDIF.

- If the *expression* evaluates as false, #ELSIF (if present) is evaluated in the same manner. If the #ELSIF *expression* is true, only the *statements* following it are generated, up to the following #ELSIF, #ELSE, or #ENDIF.
- If all preceding #IF and #ELSIF conditions evaluate false, only the *statements* following #ELSE (if present) are generated, up to the following #ENDIF. If there is no #ELSE, no code is generated.

Example:

```
#IF(SUB(%ReportControlStatement,1,6)='HEADER')
 #SET(%Indentation,%Indentation+1)
#ELSIF(SUB(%ReportControlStatement,1,6)='FOOTER')
 #SET(%Indentation,%Indentation+1)
#ELSIF(SUB(%ReportControlStatement,1,6)='DETAIL')
 #SET(%Indentation,%Indentation+1)
#ELSIF(SUB(%ReportControlStatement,1,6)='OPTION')
 #SET(%Indentation,%Indentation+1)
#ELSIF(SUB(%ReportControlStatement,1,5)='GROUP')
 #SET(%Indentation,%Indentation+1)
#ELSIF(SUB(%ReportControlStatement,1,5)='BREAK')
 #SET(%Indentation,%Indentation+1)
#ELSIF(SUB(%ReportControlStatement,1,4)='FORM')
 #SET(%Indentation,%Indentation+1)
#ENDIF
```

#IMAGE (display graphic)

#IMAGE(*picture* **)** [, **AT( )**]

#IMAGE	Displays a graphic image on a properties window.
picture	A string expression containing the name of the image file to display.
AT	Specifies the size and position of the *picture* display area in the window. This attribute takes the same parameters as the Clarion language AT attribute.

The **#IMAGE** statement displays the *picture* graphic image on a properties window. The display updates whenever the value in the *picture* changes. #IMAGE is not valid in a #MODULE section.

Example:

```
#IMAGE('SomePic.BMP')      #!Display a bitmap
```

#IMPORT(import from text script)

#IMPORT(*source* **) [,** **\| RENAME \|]**
\| REPLACE \|

#IMPORT	Creates an .APP for Clarion for Windows from a .TXA script *source* file.
source	The name of the .TXA script file from which to create the .APP file.

RENAME	Overrides the **Procedure Name Clash** prompt dialog and renames all procedures.
REPLACE	Overrides the **Procedure Name Clash** prompt dialog and replaces all procedures.

adds procedure definitions to a Clarion for Windows .APP file from a .TXA script *source* file. This is used for importing from other versions of Clarion application development products.

Example:

```
#UTILITY(SomeUtility,'Some Utility Template')
  #PROMPT('File to import',@s64),%ImportFile
  #IMPORT(%ImportFile)
```

#INCLUDE (include a template file)

#INCLUDE(*filename* **)**

#INCLUDE	Adds a template file to the Template file chain.
filename	A string constant containing the name of the template file to include.

The **#INCLUDE** statement adds a template file to the Template file chain. The template file containing the #INCLUDE statement continues to be processed after the included file has been processed.

Example:

```
#TEMPLATE(Clarion,'Clarion Standard Shipping Templates')
#INCLUDE('Clarion1.TPW')                      #!Include a template file
#INCLUDE('Clarion2.TPW')                      #!Include another template file
```

#INDENT (change indentation level)

#INDENT(*value* **)**

#INDENT	Changes the indentation level of generated code.
value	An expression that resolves to a positive or negative integer which specifies the amount and direction to change the current indentation level.

The **#INDENT** statement changes the indentation level of generated code by the amount and direction specified by the *value* parameter. If the *value* is positive, the indentation level increases (moves right). If the *value* is negative, the indentation level decreases (moves left).

Example:

```
#INDENT(-2)                  #!Change indent level left 2 spaces
```

See Also: #INSERT

#INSERT (insert code from a #GROUP)

#INSERT(*symbol* [**(** *set* **)**] [, *parameters*] **)**[, *returnvalue*] [, **NOINDENT**]

#INSERT	Inserts code from a #GROUP.
symbol	A symbol that names a #GROUP section.
set	The #TEMPLATE *name* parameter for the template set to which the #GROUP belongs. If omitted, the #GROUP must be of the same template set *name* as the #PROCEDURE in which it is used.
parameters	The parameters passed to the #GROUP. Each parameter must be separated by a comma. All parameters defined for the #GROUP must be passed; they may not be omitted.
returnvalue	A symbol to receive the value returned by the #RETURN statement.
NOINDENT	Specifies the code inserted retains the indention in its #GROUP--any source in column 1 in the #GROUP is generated into column 1, no matter where the #INSERT is placed.

The **#INSERT** statement places, at the exact position the #INSERT is located within the Template code, the code from the #GROUP named by the *symbol*. The *set* parameter specifies the #TEMPLATE that contains the #GROUP. This allows any Template to use #GROUP code from any other registered Template.

The *parameters* passed to the #GROUP fall into two categories: value-parameters and variable-parameters. Value-parameters are declared by the #GROUP as a user-defined symbol, while variable-parameters are declared by the #GROUP as a user-defined symbol with a prepended asterisk (*). Either a symbol or an expression may be passed as a value-parameter. Only a symbol may be passed as a variable-parameter.

The *returnvalue* symbol receives the value returned by the #GROUP from the #RETURN statement that terminates the #GROUP. If the #GROUP does not contain a #RETURN statement, or that #RETURN does not have a parameter, then the value received is an empty string ('').

Example:

```
#INSERT(%SomeGroup)                      #!Ordinary insert
#INSERT(%GenerateFormulas(Clarion))      #!Insert #GROUP from Clarion Template
#INSERT(%FileRecordFilter,%Secondary)    #!Insert #GROUP with passed parameter

#!#GROUP from Clarion Template with two passed parameters:
#INSERT(%FileRecordFilter(Clarion),%Primary,%Secondary)
```

See Also: #GROUP, #CALL, #RETURN

#INVOKE (insert code from a named #GROUP)

#INVOKE(*symbol* [, *parameters*] **)** [, *returnvalue*] [, **NOINDENT**]

#INVOKE	Inserts code from a named #GROUP.
symbol	A symbol containing the name of a #GROUP section (including the #TEMPLATE to which it belongs).
parameters	The parameters passed to the #GROUP. Each parameter must be separated by a comma. All parameters defined for the #GROUP must be passed; they may not be omitted.
returnvalue	A symbol to receive the value returned by the #RETURN statement.
NOINDENT	Specifies the code inserted retains the indention in its #GROUP--any source in column 1 in the #GROUP is generated into column 1, no matter where the #INSERT is placed.

The **#INVOKE** statement places, at the exact position the #INVOKE is located within the Template code, the code from the #GROUP named in the *symbol*. The *symbol* must contain the name of a #GROUP, including the #TEMPLATE set to which it belongs. The main difference between #INVOKE and #CALL is the *symbol* parameter, which is a variable containing the #GROUP to call in #INVOKE and a constant naming the #GROUP to call in #CALL.

The *parameters* passed to the #GROUP fall into two categories: value-parameters and variable-parameters. Value-parameters are declared by the #GROUP as a user-defined symbol, while variable-parameters are declared by the #GROUP as a user-defined symbol with a prepended asterisk (*). Either a symbol or an expression may be passed as a value-parameter. Only a symbol may be passed as a variable-parameter.

The *returnvalue* symbol receives the value returned by the #GROUP from the #RETURN statement that terminates the #GROUP. If the #GROUP does not contain a #RETURN statement, or that #RETURN does not have a parameter, then the value received is an empty string ('').

Example:

```
#SET(%SomeGroup,'%GenerateFormulas(Clarion)') #!Set variable to a #GROUP name
#INVOKE(%SomeGroup)

#SET(%SomeGroup,'%FileRecordFilter(Clarion)') #!Set variable to another #GROUP name
#INVOKE(%SomeGroup,%Secondary)                #!Insert #GROUP with passed parameter
#INVOKE(%SomeGroup,%Primary,%Secondary)       #!#GROUP with two passed parameters
```

See Also:

#GROUP, #INSERT, #CALL, #RETURN

#LOCALDATA (default local data declarations)

#LOCALDATA

declarations

#ENDLOCALDATA

#LOCALDATA	Begins a default local data declaration section.
declarations	Data declarations.
#ENDLOCALDATA	Terminates the default local data declarations.

The **#LOCALDATA** structure contains default data *declarations* local to the procedure generated by the #PROCEDURE procedure Template. The *declarations* are then available to the programmer through the *Data* button on the **Procedure Properties** dialog. #LOCALDATA may only be placed in a #PROCEDURE, #CODE, #CONTROL, or #EXTENSION section of the Template. The *declarations* will appear in the generated procedure between the keywords PROCEDURE and CODE.

Example:

```
#LOCALDATA
Action   BYTE         !Disk action variable
TempFile  CTRING(65)     !Temporary filename variable
#ENDLOCALDATA
```

#LOOP (iteratively generate code)

#LOOP [, | **UNTIL(** *expression* **)** |]
| **WHILE(** *expression* **)** |
| **FOR(** *counter, start, end* **)** [, **BY(** *step* **)**] |
| **TIMES(** *iterations* **)** |
statements
#ENDLOOP

#LOOP	Initiates an iterative statement execution structure.
UNTIL	Evaluates its *expression* before each iteration of the #LOOP. If its *expression* evaluates to true, the #LOOP control sequence terminates.
expression	Any Template language expression which can evaluate to false (blank or zero) or true (any other value).
WHILE	Evaluates its *expression* before each iteration of the #LOOP. If its *expression* evaluates to false, the #LOOP control sequence terminates.
FOR	Initializes its *counter* to the *start* value, and increments it by the *step* value each time through the loop. When the *counter* is greater than the *end* value, the #LOOP control sequence terminates.
counter	A user-defined symbol used as the loop counter.
start	An expression containing the initial value to which to set the loop *counter*.
end	An expression containing the ending value of the loop *counter*.
BY	Explicitly defines the increment value for the *counter*.
step	An expression containing the increment value for the *counter*. If omitted, the *step* defaults to one (1).
TIMES	Loops the number of times specified by the *iterations*.
iterations	An expression containing the number of times to loop.
statements	One or more target and/or Template Language statements.
#ENDLOOP	Terminates the #LOOP structure.

A **#LOOP** structure repetitively executes the *statements* within its structure. The #LOOP structure must be terminated by **#ENDLOOP**. If there is no #ENDLOOP, an error message is issued during Template file pre-processing. A #LOOP structure may be nested within another #LOOP structure.

The **#LOOP,UNTIL** or **#LOOP,WHILE** statements create exit conditions for the #LOOP. Their *expressions* are always evaluated at the top of the #LOOP, before the #LOOP is executed. A #LOOP WHILE structure continuously loops as long as the *expression* is true. A #LOOP UNTIL structure continuously loops as long as the *expression* is false. The *expression* may contain Template symbols, constant values, and any of the arithmetic, Boolean, and logical operators documented in the *Language Reference*. Procedure calls are allowed. If the modulus division operator (%) is used in the *expression*, it must be followed by at least one blank space (to explicitly differentiate it from the Template symbols).

The **#LOOP,FOR** statement also creates an exit condition for the #LOOP. The #LOOP initializes the *counter* to the *start* value on its first iteration. The #LOOP automatically increments the *counter* by the *step* value on each subsequent iteration, then evaluates the *counter* against the *end* value. When the *counter* is greater than the *end*, the #LOOP control sequence terminates.

#LOOP (without WHILE, UNTIL, or FOR) loops continuously, unless a #BREAK or #RETURN statement is executed. #BREAK terminates the #LOOP and continues execution with the statement following the #LOOP structure. All *statements* within a #LOOP structure are executed unless a #CYCLE statement is executed. #CYCLE immediately gives control back to the top of the #LOOP for the next iteration, without executing any statements following the #CYCLE in the #LOOP.

Example:

```
#SET(%LoopBreakFlag,'NO')
#LOOP                               #!Continuous loop
  #INSERT(%SomeRepeatedCodeGroup)
  #IF(%LoopBreakFlag = 'YES')       #!Check break condition
    #BREAK
  #ENDIF
#ENDLOOP

#SET(%LoopBreakFlag,'NO')
#LOOP,UNTIL(%LoopBreakFlag = 'YES')  #!Loop until condition is true
  #INSERT(%SomeRepeatedCodeGroup)
#ENDLOOP

#SET(%LoopBreakFlag,'NO')
#LOOP,WHILE(%LoopBreakFlag = 'NO')   #!Loop while condition is true
  #INSERT(%SomeRepeatedCodeGroup)
#ENDLOOP
```

See Also: #BREAK, #CYCLE

#MESSAGE (display source generation message)

#MESSAGE(*message, line* **)**

#MESSAGE	Displays a source generation message.
message	A string constant, or a user-defined symbol, containing a message to display in the Source Generation dialog.
line	An integer constant or symbol containing the line number on which to display the *message*. If out of the range 1 through 3, the *message* is displayed in the title bar as the window caption.

#MESSAGE displays a *message* in the Source Generation messsage dialog. The first #MESSAGE statement displays the message window. Subsequent #MESSAGE statements modify the display text.

Example:

```
#MESSAGE('Generating ' & %Application,0)   #!Display Title bar text
#MESSAGE('Generating ' & %Procedure,2)     #!Display Progress message on line 2
```

#MODULE (module area)

#MODULE(*name, description* [, *target, extension*] **)** [, **HLP(** *helpid* **)**] [, **EXTERNAL**]

#MODULE	Begins the module section.
name	The name of the Module which identifies it for the Template Registry and Template Language statements. This must be a valid Clarion label.
description	A string constant describing the #MODULE section for the Template Registry and Application Generator.
target	A string constant that specifies the source language the Template generates. The word "EXTERNAL" is convention adopted to indicate an external source or object module. If omitted, it defaults to Clarion.
extension	A string constant that specifies the source code file extension for the *target*. If omitted, it defaults to .CLW.
HLP	Specifies on-line help is available.
helpid	A string constant containing the identifier to access the Help system. This may be either a Help keyword or "context string."
EXTERNAL	Specifies no source generates into the module.

The **#MODULE** statement defines the beginning of the section of the template which puts data into each generated source module's data area. The #MODULE Section is terminated by the next Template Code Section (#PROGRAM, #MODULE, #PROCEDURE, #CONTROL, #CODE, #EXTENSION, or #GROUP) statement encountered, or the end of the file. A Template set may contain multiple #MODULE statements.

Code generated by a #MODULE section is (usually) placed at the beginning of a source code file generated by the Application Generator.

#BUTTON, #PROMPT, and #DISPLAY statements are not valid within a #MODULE section.

Example:

```
#MODULE(ExternalOBJ,'External .OBJ module','EXTERNAL','.OBJ'),EXTERNAL
#MODULE(ExternalLIB,'External .LIB module','EXTERNAL','.LIB'),EXTERNAL
#MODULE(GENERATED,'Clarion MEMBER module')
  MEMBER('%Program')           !MEMBER statement is required
%ModuleData                   #!Data declarations local to the Module
```

#OPEN (open source file)

#OPEN(*file* **)** [, **READ**]

#OPEN	Opens a disk file to receive generated source code.
file	A string constant, template symbol, or expression containing a DOS file specification. This may be a fully qualified DOS pathname.
READ	Opens the file as read-only. The file cannot be already open for output.

The **#OPEN** statement opens a disk file to receive the source code generated by #GENERATE. If the *file* does not exist, it is created. If the *file* already exists, it is opened in "append source" mode. If the *file* is already open, a source generation error is produced.The *file* is automatically selected as the active source output destination.

If the READ attribute is present, the file is opened in read-only mode so the #READ statement can read it as an ASCII text file. Only one file can be open for input at one time.

Example:

```
#SET(%ProgramFile,(%Application & '.$$$'))       #!Temp program filename
#OPEN(%ProgramFile)                              #!Open existing program file
#GENERATE(%Program)                              #!Generate main program header
#CLOSE(%ProgramFile)                             #!Close output file

#OPEN(%ProgramFile),READ                         #!Open it in read-only mode
#DECLARE(%ASCIIFileRecord)
#LOOP
  #READ(%ASCIIFileRecord)
 #! Parse the line and do something with it
 #IF(%ASCIIFileRecord = %EOF)
  #BREAK
 #ENDIF
#ENDLOOP
#CLOSE(%ProgramFile),READ
```

See Also: #READ

#CLOSE

#PDEFINE (add #pragma to project)

#PDEFINE (*symbol, value* **)**

#PDEFINE	Adds a #pragma statement to the project system.
symbol	A symbol that contains a pragma identifier.
value	The value that is associated with the symbol to determine whether to omit or compile code that is enclosed using the OMIT or COMPILE directives.

The **#PDEFINE** statement adds a #pragma (compiler option) statement to the project system. A pragma is added to the project file in the form of *symbol* => *value.*

Example:

```
#IF (%DemoVersion)
  #PDEFINE('_DEMO_',1)
#ELSE
  #PDEFINE('_DEMO_',0)
#ENDIF
```

#POP (delete and re-fix a multi-value symbol)

#POP(*symbol* **)**

#POP	Deletes the last instance of a multi-valued symbol and re-FIXes to the new last instance.
symbol	A multi-valued user-defined symbol.

The **#POP** statement deletes the last instance of a multi-valued symbol and re-FIXs to the new last instance. This is directly equivalent to issuing a #DELETE(%MultiSymbol,ITEMS(%MultiSymbol)) statement followed by #SELECT(%MultiSymbol,ITEMS(%MultiSymbol)) statement.

Example:

```
#POP(%MultiSymbol)  #!Delete last instance and fix to new last instance
```

#POSTEMBED (generate ending embed point comments)

#POSTEMBED(*text* [*, condition*] **)**

#POSTEMBED	Generates comments at the end of embed point code.
text	A string constant or constant expression containing the text to place in the embed point.
condition	An expression that, when true, allows the comments to generate.

The **#POSTEMBED** statement specifies that comments generate at the end of embed points that contain code. The output *condition* is usually the value of a global prompt.

The comment *text* may use the %EmbedID, %EmbedDescription, and %EmbedParameters built-in symbols to identify the embed point:

%EmbedID	The current embed point's identifying symbol.
%EmbedDescription	The current embed point's description.
%EmbedParameters	The current embed point's current instance, as a comma-delimited list.

Example:

```
#POSTEMBED('! After Embed Point: ' & %EmbedID & ' ' & %EmbedDescription & ' ' & |
              %EmbedParameters,%GenerateEmbedComments)
```

See Also:	#PREEMBED
	#EMPTYEMBED

#PREEMBED (generate beginning embed point comments)

#PREEMBED(*text* [*, condition*] **)**

#PREEMBED	Generates comments at the beginning of embed point code.
text	A string constant or constant expression containing the text to place in the embed point.
condition	An expression that, when true, allows the comments to generate.

The **#PREEMBED** statement specifies that comments generate at the beginning of embed points that contain code. The output *condition* is usually the value of a global prompt.

The comment *text* may use the %EmbedID, %EmbedDescription, and %EmbedParameters built-in symbols to identify the embed point:

%EmbedID	The current embed point's identifying symbol.
%EmbedDescription	The current embed point's description.
%EmbedParameters	The current embed point's current instance, as a comma-delimited list.

Example:

```
#PREEMBED('! Before Embed Point: ' & %EmbedID & ' ' & %EmbedDescription & ' ' & |
           %EmbedParameters,%GenerateEmbedComments)
```

See Also: #POSTEMBED

#EMPTYEMBED

#PREPARE (setup prompt symbols)

#PREPARE

statements

#ENDPREPARE

#PREPARE	Begins a prompts symbol setup section.
statements	Template language statements to fix multi-valued symbols to the values needed to process the #PROMPT or #BUTTON statement preceding the #PREPARE.
#ENDPREPARE	Terminates the section.

The **#PREPARE** structure contains Template language statements to fix multi-valued symbols to the values needed to process the #PROMPT or #BUTTON statements preceding the #PREPARE.

A #PREPARE section that precedes all the prompts is executed once, before the prompts are added to a window. In particular, if the prompts are displayed on the procedure properties screen, editing the window definition and returning does not execute the prepare code.

Example:

```
#CODE (SetProperty, 'Set a property on a control')
#PREPARE
  #DECLARE(%Choices),MULTI
  #DECLARE(%NextLine)
  #FREE(%Choices)
  #OPEN('property.clw'),READ
  #READ(%NextLine)
  #LOOP WHILE(%NextLine <> %Eof)
    #! Exclude PROPLIST: properties
    #IF (SUB(%NextLine, 1, 5) = 'PROP:')
      #ADD(%Choices, SUB(%NextLine, 1, INSTRING(' ',%NextLine)-1))
    #END
    #READ (%NextLine)
  #END
  #CLOSE,READ
#ENDPREPARE
#PROMPT('Control:', FROM(%Control)),%Target
#PROMPT('Property:', FROM(%Choices)),%Selection
#PROMPT('Value:', @s255),%TargetValue
%target{%Selection} = %TargetValue
```

#PRINT (print a source file)

#PRINT(*file, title* **)**

#PRINT	Prints a *file* to the current Windows printer.
file	A string constant, template symbol, or expression containing a DOS file specification. This may be a fully qualified DOS pathname.
title	A string constant, template symbol, or expression containing the title to generate for the *file*.

The **#PRINT** statement prints the contents of the *file* to the Windows default printer.

Example:

```
#FOR(%Module)
  #SET(%ModuleFile,(%Module & '.CLW'))           #!Set to existing module file
  #PRINT(%ModuleFile,"Printout ' & %ModuleFile)
#ENDFOR
```

#PRIORITY (set new embed priority level)

#PRIORITY(*number* **)** [,**DESCRIPTION**(*treedesc*)]

#PRIORITY	Specifies a new priority level for the code following the #PRIORITY statement.
number	An integer constant in the range 1 to 10000.
DESCRIPTION	An optional description of the priority point.
treedesc	A string constant or expression that will appear in the embed tree for this priority point.

The **#PRIORITY** statement specifies a new priority *number* for the code statements following the #PRIORITY.

#PRIORITY is only valid when used in the #AT/#ENDAT structure. It cannot exist within other nested structures contained within the #AT/#ENDAT structure (e.g., #PRIORITY cannot be placed inside of a #CASE/'#ENDCASE which is nested within a #AT/#ENDAT).

Example:

```
#CONTROL(BrowseList,'Add Browse List controls')
  #AT(%ControlEvent,'?Insert','Accepted'),PRIORITY(5000)  #!Start at priority 5000
GlobalRequest = InsertRecord                              #!Goes into priority 5000
  #PRIORITY(8500),DESCRIPTION('Before calling update procedure')

%UpdateProc                                               #!Goes into priority 8500
  #ENDAT
#!
```

See Also: #AT (insert code in an embed point)

#EMBED

#CODE

#CONTROL

#EXTENSION

#PROCEDURE (begin a procedure template)

#PROCEDURE(*name, description* [, *target*] **)** [, **REPORT**] [, **WINDOW**] [, **HLP(** *helpid* **)**]

[, **PRIMARY(** *message* [**,** *flag*] **)**] [, **QUICK(** *wizard* **)**]

#PROCEDURE	Begins a procedure template.
name	The label of the procedure template. This must be a valid Clarion label.
description	A string constant describing the procedure Template.
target	A string constant that specifies the source language the template generates. If omitted, it defaults to Clarion.
REPORT	Tells the Application Generator to make the Report Formatter available.
WINDOW	Tells the Application Generator to make the Window Formatter available.
HLP	Specifies on-line help is available.
helpid	A string constant containing the help identifier. This may be either a Help keyword or "context string."
PRIMARY	Specifies at least one file must be placed in the procedure's File Schematic.
message	A string constant containing a message that appears in the File Schematic next to the procedure's Primary file.
flag	If present, contains OPTIONAL (the file is not required), OPTKEY (the key is not required), or NOKEY (the file is not required to have a key).
QUICK	Specifies the procedure has a wizard #UTILITY that runs when the **Use Procedure Wizard** box is checked.
wizard	The identifier (including template class, if necessary) of the wizard #UTILITY template.

The **#PROCEDURE** statement begins a Procedure template. A Procedure template contains the Template and *target* language statements used to generate the source code for a procedure within your application. A #PROCEDURE section is terminated by the first occurrence of a Template Code Section (#PROGRAM, #MODULE, #PROCEDURE, #CONTROL, #CODE, #EXTENSION, or #GROUP) statement, or the end of the file. Within a Template set you may have multiple #PROCEDURE sections, but they must all have unique *name* parameters.

Example:

```
#PROCEDURE(ProcName1,'This is a sample window procedure'),WINDOW
#PROCEDURE(ProcName2,'This is a sample report procedure'),REPORT
#PROCEDURE(ProcName3,'This is a sample anything procedure'),WINDOW,REPORT
#PROCEDURE(Browse,'List with Wizard'),WINDOW,QUICK(BrowseWizard(Wizards))
```

#PROGRAM (global area)

#PROGRAM(*name, description* [, *target, extension*] **)** [, **HLP(** *helpid* **)**]

#PROGRAM	Defines the beginning of the main program module.
name	The name of the #PROGRAM which identifies it for the Template Registry and Template Language statements. This must be a valid Clarion label.
description	A string constant describing the #PROGRAM section for the Template Registry and Application Generator.
target	A string constant that specifies the source language the Template generates. If omitted, it defaults to Clarion.
extension	A string constant that specifies the source code file extension for the *target*. If omitted, it defaults to .CLW.
HLP	Specifies on-line help is available.
helpid	A string constant containing the identifier to access the Help system. This may be either a Help keyword or "context string."

The **#PROGRAM** statement defines the beginning of the main program module of the Template. The #PROGRAM section is terminated by the next Template Code Section (#MODULE, #PROCEDURE, #CONTROL, #CODE, #EXTENSION, or #GROUP) statement encountered, or the end of the file. Only one #PROGRAM section is allowed in a Template set.

#BUTTON, #PROMPT, and #DISPLAY statements are not valid within a #PROGRAM section. Global prompts go in the #APPLICATION section.

Example:

```
#PROGRAM(CLARION,'Standard Clarion Shipping Template')
  PROGRAM                  !PROGRAM statement required
  INCLUDE('Keycodes.clw')
   INCLUDE('Errors.clw')
   INCLUDE('Equates.clw')
```

#PROJECT (add file to project)

#PROJECT(*module* **)**

#PROJECT	Includes a source or object code library, or Project file, in the application's Project file.
module	A string constant which names a source (.CLW, if Clarion is the target language), object (.OBJ), or library (.LIB) file containing procedures required by the procedure Template. This may also name a Project (.PRJ) file to be called by the application's Project. The type of file being imported is determined by the file extension.

The **#PROJECT** statement specifies a source or object code library, or Project file, which is required to be in the application for the correct functioning of procedures created by the procedure Template.

#PROJECT provides a direct method of communicating *module* information to the Application Generator and Project system. It alerts the Application Generator to the required presence of the *module* for compiling and/or linking the application. Therefore, the application's Project file (generated by the Application Generator) automatically includes the *module* for making, compiling, and/or linking.

If multiple instances of the same #PROJECT statement are referenced by procedures created in the application, only the first is used. This would occur when multiple procedure Templates require the same *module*, or multiple application procedures are created from the same procedure Template.

#PROJECT allows a developer to automate the installation of third-party libraries and Templates to other developer's computers. This ensures that the application's Project is generated correctly.

#PROJECT for a Project (.PRJ) file provides the ability to create a hierarchy of Projects on large development projects. Where multiple libraries are being linked into a package, this allows you to ensure "make dependencies" are met for all libraries referenced in a particular project.

Example:

```
#AT(%CustomGlobalDeclarations)
  #PROJECT('Party3.LIB')
#ENDAT
```

#PROMPT (prompt for programmer input)

#PROMPT(*string*, *type* **)** [, *symbol*] [, **REQ**] [, **DEFAULT(** *default* **)**] [, **ICON(** *file* **)**] [, **AT( )**]

[, **PROMPTAT( )**] [, **MULTI(** *description* **)**] [, **UNIQUE**] [, **INLINE**] [, **VALUE(** *value* **)**]
[, **SELECTION(** *description* **)**] [, **CHOICE**] [, **WHENACCEPTED(** *expression* **)**]
[, **HSCROLL**] [, **PROP(** *name,value* **)**]

#PROMPT	Asks the programmer for input.
string	A string constant containing the text to display as the input prompt. This may contain an ampersand (&) denoting a "hot" key used in conjunction with the ALT key to get to this field on the properties screen.
type	A picture token or prompt keyword.
symbol	A User-defined symbol to receive the input. A #PROMPT with a RADIO or EMBED *type* cannot have a *symbol*, all other *types* must have a *symbol*.
REQ	Specifies the prompt cannot be left blank or zero.
DEFAULT	Specifies an initial value (which may be overridden).
default	A string constant containing the initial value.
ICON	Specifies an icon for the button face of a #PROMPT with the MULTI attribute.
file	A string constant containing the name of the .ICO file to display on the button face.
AT	Specifies the position of the prompt entry area in the window, relative to the first prompt placed on the window from the Template (excluding the standard prompts on every procedure properties window). This attribute takes the same parameters as the Clarion language AT attribute.
PROMPTAT	Specifies the position of the prompt *string* in the window, relative to the first prompt placed on the window from the Template (excluding the standard prompts on every procedure properties window). This attribute takes the same parameters as the Clarion language AT attribute.
MULTI	Specifies the programmer may enter multiple values for the #PROMPT. The prompt appears as a button which pops up a list box allowing the programmer to enter multiple values, unless the INLINE attribute is also present.
description	A string constant containing the name to display on the button face and at the top of the list of prompt values.
UNIQUE	The multiple values the programmer enters for the #PROMPT are unique values and are sorted in ascending order. The MULTI attribute must also be present.
INLINE	The multiple values the programmer enters for the #PROMPT appears as a list box with update buttons which allow the programmer to enter multiple values. The MULTI attribute must also be present.

VALUE	Specifies the value assigned to the *symbol* when the #PROMPT is selected. Valid only when the #PROMPT *type* is RADIO.
value	A string constant containing the value to assign to the *symbol*.
SELECTION	Specifies the programmer may select multiple values for the #PROMPT from the list of choices presented by the FROM *type*. The prompt appears as a button which pops up a list box allowing the programmer to choose multiple values, unless the INLINE attribute is also present.
CHOICE	Specifies the *symbol* receives the ordinal position number of the selection instead of the *string* text from a DROP or OPTION/RADIO #PROMPT set.
WHENACCEPTED	Specifies an *expression* to execute when the #PROMPT posts an Accepted event (the programmer has entered or chosen data).
group	The name of the #GROUP to execute.
HSCROLL	Specifies horizontal scroll bars in the DROP list. Valid only when the #PROMPT *type* is DROP.
PROP	Specifies a property to assign to the prompt text. *Name* designates the property name equate (Example: PROP:FontColor) and *value* is the value assigned to the named property (Example: 0FFFFFFH)

The **#PROMPT** statement asks the programmer for input. A #PROMPT statement may be placed in #APPLICATION, #PROCEDURE, #CODE, #CONTROL, #EXTENSION, #UTILITY, or #FIELD sections. It may not be placed in a #PROGRAM, #MODULE, #TEMPLATE, or #GROUP section.

When the #PROMPT is placed in a template section, the prompt *string* and its associated entry field are placed as follows:

Section Name	Window Name
#APPLICATION	Global Settings
#PROCEDURE	Procedure Properties
#CODE	Embeds Dialog
#CONTROL	Control Properties Actions Tab
#EXTENSION	Extensions Dialog
#FIELD	Control Properties Actions Tab

The *type* parameter may either contain a picture token to format the programmer's input, or one of the following keywords:

PROCEDURE	The label of a procedure
FILE	The label of a data file
KEY	The label of a key (can be limited to one file)
COMPONENT	The label of a key component field (can be limited to one key)
FIELD	The label of a file field (can be limited to one file)
EXPR	A multi-field selection box that builds an expression
OPTFIELD	Constant text or the label of a file field
FORMAT	Calls the listbox formatter.
PICTURE	Calls the picture token formatter.
DROP	Creates a droplist of items specified in its parameter
KEYCODE	A keycode or keycode EQUATE
OPTION	Creates a radio button structure
RADIO	Creates a radio button
CHECK	Creates a check box
CONTROL	A window control
FROM	Creates a droplist of items contained in its symbol parameter
EMBED	Allows the user to edit a specified embedded source code point
SPIN	Creates a spin control
TEXT	Creates a text entry control
OPENDIALOG	Calls a standard Windows Open File dialog
SAVEDIALOG	Calls a standard Windows Save File dialog
COLOR	Calls a standard Windows Color dialog

For all *types* except RADIO and CHECK (and MULTI attribute prompts), the #PROMPT *string* is displayed on the screen immediately to the left of its data input area. A #PROMPT with the REQ attribute cannot be left blank or zero; it is a required input field. The DEFAULT attribute may be used to provide the programmer with an initial *value* in the #PROMPT, which may be overidden at design time.

A #PROMPT with a RADIO *type* creates one Radio button for the immediately preceding #PROMPT with an OPTION *type*. There may be multiple RADIOs for one OPTION. Each RADIO's *string*, when selected, is placed in the closest preceding OPTION's *symbol*. The OPTION structure is terminated by the first #PROMPT following it that is not a RADIO.

The MULTI attribute specifies the programmer may enter multiple values for the #PROMPT. A button appears on the Properties window with the *description* on its face. Alternatively, this can have an ICON attribute to name an .ICO file to display on the button face. This button calls a window containing a list box to display all the multiple values entered for the #PROMPT, along with Insert, Change, and Delete buttons. These three buttons call another window containing the #PROMPT *string* and its data entry field to allow the programmer to update the entries in the list.

When the programmer has entered a value for the #PROMPT, the input value is assigned to the *symbol*. The value entered by the programmer may be checked for validity by one or more #VALIDATE statements immediately following the #PROMPT statement.

The value(s) placed in the *symbol* may be used or evaluated elsewhere within the Template. A *symbol* defined by a #PROMPT in the #APPLICATION section of the Template is Global, it can be used or evaluated anywhere in the Template. A *symbol* defined by #PROMPT in a #PROCEDURE section is Local, and is a dependent symbol to %Procedure; it can be used or evaluated only within that #PROCEDURE section. A *symbol* defined by #PROMPT in a #CODE, #CONTROL, or #EXTENSION section of the Template can be used or evaluated only within that section.

Example:

```
#PROMPT('Ask for Input',@s20),%InputSymbol        #!Simple input
#PROMPT('Ask for FileName',FILE),%InputFile,REQ   #!Required filename
#PROMPT('Pick One',OPTION),%InputChoice           #!Mutually exclusive options
#PROMPT('Choice One',RADIO)
#PROMPT('Choice Two',RADIO)
#PROMPT('Next Procedure',PROCEDURE),%NextProc     #!Prompt for procedure name
                                                  #!Prompt for multiple input:
#PROMPT('Ask for Multiple Input',@s20),%MultiSymbol,MULTI('Input Values...')
#PROMPT('ABC Version',@S10),%ABCVersion,DEFAULT('6000'),PROP(PROP:READONLY,1) #!Display
```

See Also: #DISPLAY

#VALIDATE

#GROUP

#BOXED

#ENABLE

#BUTTON

#PROMPT Entry Types Alpha Listing

CHECK (check box)

CHECK

The **CHECK** *type* in a #PROMPT statement indicates the prompt's *symbol* is a toggle switch which is used only for on/off, yes/no, or true/false evaluation. CHECK puts a check box on screen in the entry area for the #PROMPT. When the Check box is toggled on, the prompt's *symbol* equals %True. When the Check box is toggled off, the prompt's *symbol* equals %False.

Example:

```
#PROMPT('Network Application',CHECK),%NetworkApp
```

COLOR (call Color dialog)

COLOR

The **COLOR** *type* in a #PROMPT statement indicates the prompt's *symbol* must contain the name of the color selected by the programmer from the Windows standard Color dialog.

Example:

```
#PROMPT('Ask for Color',COLOR),%ColorSymbol
```

COMPONENT (list of KEY fields)

COMPONENT [**(** *scope* **)**]

COMPONENT	Displays a list of KEY component fields.
scope	A symbol containing a KEY. If omitted, the list displays all KEY components for all KEYs in all FILEs.

The **COMPONENT** *type* in a #PROMPT statement indicates the prompt's *symbol* must contain the label of one of the component fields of a KEY. A list of available KEY fields pops up when the #PROMPT is encountered on the Properties screen.

The COMPONENT may have a *scope* parameter that limits the KEY components available in the list. If *scope* is the label of a KEY, the list displays all KEY components for that KEY.

Example:

```
#PROMPT('Record Selector',COMPONENT(%Primary)),%RecordSelector
```

CONTROL (list of window fields)

CONTROL

The **CONTROL** *type* in a #PROMPT statement indicates the prompt's *symbol* must contain the field equate label of a window control. A list of available controls pops up when the #PROMPT is encountered on the Properties screen.

Example:

```
#PROMPT('Locator Field',CONTROL),%Locator
```

DROP (droplist of items)

DROP [**(** *scope* **)**]

DROP	Creates a droplist of items.
scope	A string constant or expression containing the items for the list. Each item must be delimited by the vertical bar (|) character.

The **DROP** *type* in a #PROMPT statement indicates the prompt's *symbol* must contain one item from the list specified in the *scope* parameter.

The *scope* parameter must contain all the items for the list. The *scope* may optionally contain text in square brackets following each choice that specifies the value to assign to the #PROMPT's *symbol*. This is useful for internationalization.

The list of items drops down just like a Clarion language LIST control with the DROP attribute. If no default value is specified, the prompt's symbol defaults to the first value in the *scope* list.

Example:

```
#PROMPT('If file not found',DROP('Create the file|Halt Program')),%FileNotFound

#PROMPT('Que es esto?',DROP('Nombre[Name]|Numero[Number]')),%WhatIsThis
  #!"Name" is the value put in %WhatIsThis when the user chooses "Nombre"
  #!"Number" is the value put in %WhatIsThis when the user chooses "Numero"
```

EMBED (enter embedded source)

EMBED(*identifier* [, *instance*] **)**

EMBED	Specifies the prompt directly edits an embedded source code point.
identifier	The user-defined template symbol which identifies the #EMBED embedded source code point to edit.
instance	A string constant or expression containing one of the values in the multi-valued symbol used by the #EMBED. You must have as many *instances* as are necessary to explicitly identify the single #EMBED point instance to edit.

The **EMBED** *type* in a #PROMPT statement indicates the prompt is used to directly edit an embedded source code point. This places an an entry area with an ellipsis (...) button next to the prompt to allow the user access to the embedded source code point. The programmer may enter a procedure call in the entry area, or press the ellipsis (...) button to go into the normal source dialog.

If the #EMBED is associated with a multi-valued symbol, you must identify the specific *instance* of the #EMBED. If you use a multi-valued symbol as an *instance* expression, it must be fixed to a single value. Most commonly, this would be used in a #FIELD structure.

Example:

```
#PROMPT('Embedded Data Declarations',EMBED(%DataSection))
#FIELD, WHERE(%ControlType = 'BUTTON')
  #PROMPT('Action when button is pressed',EMBED(%ControlEvent,%Control,'Accepted'))
#ENDFIELD
```

EMBEDBUTTON (call Embeds dialog)

EMBEDBUTTON(*identifier* [,*instance1,…instanceN*] **)**

identifier	The user-defined template symbol which identifies the embedded source code point to edit.
instance	A string constant or expression containing one of the values in the multi-valued symbol used by the embedded dialog opened by . You must have as many *instances* as are necessary to explicitly identify the single #EMBED point instance to edit.

The **EMBEDBUTTON** *type* in a #PROMPT statement indicates the prompt's *symbol* will used to produce text on a button that opens the embedded source dialog. The embed points displayed are determined by the identifier's template symbol and optional instances used to distinguish different embed areas. Essentially, this prompt allows the EMBED functionality to be called from a secondary window, eliminating screen clutter.

Example:

```
#PROMPT('Advanced...',EMBEDBUTTON(%ControlPostEventHandling,%Control,'Selected'))
```

EXPR (appended data fields)

EXPR

The **EXPR** *type* in a #PROMPT statement indicates the prompt's *symbol* will contain an expression concatenating multiple fields in a data file. When a field is selected, the field name is appended to the existing *symbol* contents (Field1 & Field2 & Field3). A list of available fields pops up when the programmer presses the "..." button next to the #PROMPT entry box on the Properties screen.

Example:

```
#PROMPT('Default Value',EXPR),%DefaultValue
```

FIELD (list of data fields)

FIELD [**(** *scope* **)**]

FIELD	Displays a list of fields in FILEs.
scope	A symbol containing a FILE label. If omitted, the list displays all fields for all FILEs.

The **FIELD** *type* in a #PROMPT statement indicates the prompt's *symbol* must contain the label of a field in a data file. A list of available fields pops up when the #PROMPT is encountered on the Properties screen.

There may be a *scope* parameter that limits the fields available in the list. If *scope* names a FILE, the list displays all fields in the FILE. If there is no *scope* parameter, the list displays all fields in all FILEs.

Example:

```
#PROMPT('Locator Field',FIELD(%Primary)),%Locator
```

FILE (list of files)

FILE

The **FILE** *type* in a #PROMPT statement indicates the prompt's *symbol* must contain the label of a data file. A list of available files from the procedure's File Schematic pops up when the #PROMPT is encountered on the Properties screen.

Example:

```
#PROMPT('Logout File',FILE),%LogoutFile
```

FORMAT (call listbox formatter)

FORMAT

The **FORMAT** *type* in a #PROMPT statement indicates the prompt's *symbol* must contain a LIST or COMBO control's FORMAT attribute string, so it calls the listbox formatter to create it.

Example:

```
#PROMPT('Alternate LIST format',FORMAT),%AlternateFormatString
```

FROM (list of symbol values)

FROM(*symbol* [, *expression*] [, *value*] **)**

FROM	Specifies a drop-down list of values from the *symbol*.
symbol	A multi-valued symbol (built-in or user-defined).
expression	An expression which controls which *symbol* values are displayed. Only *symbol* values where the *expression* is true are displayed in the drop list.
value	The symbol containing the values to display for the prompt and assigned mto the *symbol*.

The **FROM** *type* in a #PROMPT statement indicates the user must select one item from the list contained in the *symbol*. The *expression* can be used to limit the *values* displayed, while the *value* defines the display elements.

Example:

```
#PROMPT('Select an Event',FROM(%ControlEvent)),%WhichEvent
#PROMPT('Select a Button',FROM(%ControlField,%ControlType = 'BUTTON')),%WhichButton
#PROMPT('Pick a Field',FROM(%Control,%ControlUse <> '',%ControlUse)),%MyButton
```

KEY (list of keys)

KEY [**(** *scope* **)**]

KEY	Displays a list of KEYs.
scope	A symbol containing a FILE. If omitted, the list displays all KEYs in all FILEs.

The **KEY** *type* in a #PROMPT statement indicates the prompt's *symbol* must contain the label of a KEY. A list of available keys from the data dictionary pops up when the #PROMPT is encountered on the Properties screen.

There may be a *scope* parameter that limits the KEYs available in the list. If *scope* names a FILE, the list displays all KEYs in the FILE. If there is no *scope* parameter, the list displays all KEYs in all FILEs.

Example:

```
#PROMPT('Which Key',KEY(%Primary)),%UseKey
```

KEYCODE (list of keycodes)

KEYCODE

The **KEYCODE** *type* in a #PROMPT statement indicates the prompt's *symbol* must contain a keycode or keycode equate label. A selection list of keycode equate labels from KEYCODES.EQU pops up when the user presses the ellipsis button next to the prompt on the Properties screen.

Example:

```
#PROMPT('Hot Key',KEYCODE),%ActiveKey
```

OPENDIALOG (call Open File dialog)

OPENDIALOG(*title, extensions* **)**

OPENDIALOG	Calls the Windows common Open File dialog.
title	A string constant containing the title to place on the file open dialog.
extensions	A string constant containing the available file extension selections for the List Files of Type drop list.

The **OPENDIALOG** *type* in a #PROMPT statement indicates the prompt's *symbol* must contain the name of the file selected by the programmer from the Open File dialog (see FILEDIALOG in the *Language Reference*). The file must already exist on disk.

Example:

```
#PROMPT('Ask for File',OPENDIALOG('Pick File','Source|*.CLW')),%FileSymbol
```

OPTFIELD (optional text or data field)

OPTFIELD

The **OPTFIELD** *type* in a #PROMPT statement indicates the prompt's *symbol* may either contain text entered by the programmer or the label of a field in a data file. When a field is selected, the #PROMPT symbol contains the field name with an exclamation prepended (!FieldName). A list of available fields pops up when the programmer presses the "..." button next to the #PROMPT entry box on the Properties screen.

Example:

```
#PROMPT('Default Value',OPTFIELD),%DefaultValue
```

OPTION (display radio buttons)

OPTION

The **OPTION** *type* in a #PROMPT statement indicates the prompt's *symbol* receives the value of one of the *strings* in one of the following RADIO #PROMPT statements, unless the CHOICE attribute is present, then the *symbol* receives the ordinal position number of the RADIO #PROMPT the programmer chooses from an OPTION set instead of the *string* text.Each of the *strings* displays a radio button on the Properties screen when the #PROMPT is encountered.

Example:

```
#PROMPT('Ask for Choice',OPTION),%OptionSymbol
#PROMPT('Option One',RADIO)
#PROMPT('Option Two',RADIO)
#PROMPT('Option Three',RADIO)
```

PICTURE (call picture formatter)

PICTURE

The **PICTURE** *type* in a #PROMPT statement calls the picture formatter to create a picture token used to format data for display.

Example:

```
#PROMPT('Display Format',PICTURE),%DisplayPicture
```

PROCEDURE (add to logical procedure tree)

PROCEDURE

The **PROCEDURE** *type* in a #PROMPT statement indicates the value placed in the *symbol* is the name of a procedure in your application. This procedure name is added to the Application Generator's logical procedure call tree in the appropriate place.

Example:

```
#PROMPT('Next Procedure',PROCEDURE),%NextProcedure
```

RADIO (one radio button)

RADIO

The **RADIO** *type* in a #PROMPT statement creates one RADIO button for the closest preceding OPTION prompt. When selected, the RADIO's *string* is placed in the OPTION's *symbol* unless the CHOICE attribute or VALUE is present. With the CHOICE attribute on the OPTION, the *symbol* receives the ordinal position number of the RADIO #PROMPT the programmer chooses from an OPTION set instead of the *string* text. With the VALUE attribute on the RADIO, the *symbol* receives *value* text.

Example:

```
#PROMPT('Ask for Choice',OPTION),%OptionSymbol
#PROMPT('Option One',RADIO)
#PROMPT('Option Two',RADIO)
#PROMPT('Option Three',RADIO)

#PROMPT('Ask for Another Choice',OPTION),%OptionSymbol2,CHOICE
#PROMPT('Option A',RADIO)                   #!%OptionSymbol2 receives 1
#PROMPT('Option B',RADIO)                   #!%OptionSymbol2 receives 2
#PROMPT('Option C',RADIO)                   #!%OptionSymbol2 receives 3

#PROMPT('Ask for Yet Another Choice',OPTION),%OptionSymbol3
#PROMPT('Option A',RADIO),VALUE('A')        #!%OptionSymbol3 receives A
#PROMPT('Option B',RADIO),VALUE('B')        #!%OptionSymbol3 receives B
#PROMPT('Option C',RADIO),VALUE('C')        #!%OptionSymbol3 receives C
```

SAVEDIALOG (call Save File dialog)

SAVEDIALOG(*title, extensions* **)**

SAVEDIALOG	Calls the Windows common Save File dialog.
title	A string constant containing the title to place on the file save dialog.
extensions	A string constant containing the available file extension selections for the List Files of Type drop list.

The **SAVEDIALOG** *type* in a #PROMPT statement indicates the prompt's *symbol* must contain the name of the file selected by the programmer from the Save File dialog (see FILEDIALOG in the *Language Reference*). The file must not already exist on disk.

Example:

```
#PROMPT('Ask for FileName',SAVEDIALOG('Pick File','Source|*.CLW')),%FileSymbol
```

SPIN (spin box)

SPIN(*picture, low, high* [, *step*] **)**

SPIN	Creates a spin control.
picture	A data entry picture token.
low	A numeric constant or expression containing the lowest valid value.
high	A numeric constant or expression containing the highest valid value.
step	A numeric constant or expression containing the amount to change each increment between lowest and highest valid values. If omitted, the default is 1.

The **SPIN** *type* in a #PROMPT statement creates a spin control for the programmer to select a valid number.

Example:

```
#PROMPT('How Many?',SPIN(@n2,1,10)),%SpinSymbol
```

TEXT (text box)

TEXT

The **TEXT** *type* in a #PROMPT statement creates a multi-line text entry control for the programmer to enter text into.

Example:

```
#PROMPT('Comment?',TEXT),%Comments
```

#PROTOTYPE (procedure prototype)

#PROTOTYPE(*parameter list* **)**

#PROTOTYPE	Assigns the *parameter list* to the **Prototype** entry field.
parameter list	A string constant containing the procedure's prototype parameter list (the entire procedure prototype without the leading procedure name) for the application's MAP structure (see the discussion of *Procedure Prototypes* in the *Language Reference*).

The **#PROTOTYPE** statement assigns the *parameter list* to the **Prototype** entry field on the Application Generator's Procedure Properties window, which automatically "dims out" the field (the programmer may not override this Prototype). This allows you to create procedure Templates which require a specific parameter list without forcing the programmer to know the procedure's prototype.

The #PROTOTYPE statement is valid only within a #PROCEDURE section and only one is allowed per #PROCEDURE section. If there is no #PROTOTYPE statement in the #PROCEDURE, the programmer is allowed to change it.

Example:

```
#PROCEDURE(SomeProc,'Some Procedure Template')
%Procedure PROCEDURE(Parm1,Parm2,Parm3)
  #PROTOTYPE('(STRING,*LONG,<*SHORT>)')
         #!This procedure expects three parameters:
         #!  a STRING passed by value
         #!  a LONG passed by address
         #!  a SHORT passed by address which may be omitted

#PROCEDURE(SomeFunc,'Some Template Procedure')
%Procedure PROCEDURE(Parm1,Parm2,Parm3)
  #PROTOTYPE('(STRING,*LONG,<*SHORT>),STRING')
         #!This procedure expects three parameters:
         #!  a STRING passed by value
         #!  a LONG passed by address
         #!  a SHORT passed by address which may be omitted
         #!It returns a STRING
```

#PURGE (delete all single or multi-valued symbol instances)

#PURGE(*symbol* **)**

#PURGE	Deletes the values from all instances of a user-defined symbol.
symbol	A user-defined symbol.

The **#PURGE** statement deletes all values from the *symbol*. If there are any symbols dependent upon the *symbol*, they are also cleared. If the *symbol* is dependent upon a multi-valued symbol, all instances of that dependent *symbol* are purged for all instances of the symbol upon which it is dependent.

Example:

```
#DECLARE(%ProcFilesPrefix),MULTI        #!Declare multi-valued symbol
#ADD(%ProcFilesPrefix,'SAV')            #!Add a value
#ADD(%ProcFilesPrefix,'BAK')            #!Add a value
#ADD(%ProcFilesPrefix,'PRE')            #!Add a value
#ADD(%ProcFilesPrefix,'BAK')            #!Add a value
#ADD(%ProcFilesPrefix,'QUE')            #!Add a value
#!%ProcFilesPrefix contains: SAV, BAK, PRE, BAK, QUE
#PURGE(%ProcFilesPrefix)                #!Delete all instances
```

See Also:	#DECLARE #ADD

#QUERY (conditionally generate source)

#QUERY(*section, source* **)**

#QUERY	Generates *source* if the named #SUSPEND *section* has been released.
section	The name of the #SUSPEND section to #QUERY.
source	The code to generate.

The **#QUERY** statement conditionally generates its *source* based on whether the named #SUSPEND *section* has been released (either explicitly or implicitly). If the #SUSPEND *section* has been released then #QUERTY generates its source, without affecting the release of any other nested #SUSPEND section it may be in--it does not create an implict #RELEASE.

Example:

```
#SUSPEND(Fred)       #!Begin suspended generation section named "Fred"
#?1
#SUSPEND             #!Begin unnamed suspended generation section
#?2
#QUERY(Fred,'3')     #!Generate "3" only if the "Fred" section is released
#?4
#RESUME
5                    #!Unconditional generation causes implicit #RELEASE of "Fred"
#RESUME              #!End "Fred" section

 #!The above code will generate 1, 3, 5 on successive output lines
 #!The 2 and 4 don't generate because the unnamed section was not released
```

See Also: #SUSPEND

#READ (read one line of a source file)

#READ(*symbol* **)**

#READ	Reads the next record from the opened read-only file.
symbol	The symbol to receive the text from the file.

The **#READ** statement reads the next record (up to the next CR/LF encountered) from open read-only file. The *symbol* receives the text from the file. If the last record has been read, the *symbol* will contain a value equivalent to the %EOF built-in symbol.

Example:

```
#OPEN(%ProgramFile),READ              #!Open it in read-only mode
#DECLARE(%ASCIIFileRecord)
#LOOP
  #READ(%ASCIIFileRecord)
   #! Parse the line and do something with it
 #IF(%ASCIIFileRecord = %EOF)
  #BREAK
 #ENDIF
#ENDLOOP
#CLOSE(%ProgramFile),READ             #!Close the read-only file
```

See Also: #OPEN

#CLOSE

#REDIRECT (change source file)

#REDIRECT([*file*] **)**

#REDIRECT	Changes the current generated source destination file.
file	A string constant, template symbol, or expression containing a DOS file specification that has already been opened with #OPEN or #CREATE. This may be a fully qualified DOS pathname. If omitted, the source generation destination returns to the previous file that received generated source.

The **#REDIRECT** statement changes the destination *file* for generated source code. All source generation output is directed to the specified *file* until a #OPEN or another #REDIRECT statement is executed. If the file has not been previously opened (or created), or is closed, then a source generation error is produced.

The destination files for generated source are kept as a LIFO (Last In, First Out) "stack" list. When #REDIRECT is issued without a *file* parameter, the source generation destination reverts to the previous destination file.

Example:

```
#SET(%NewProgramFile,(%Application & '.CLW'))    #!Temp new program filename
#CREATE(%NewProgramFile)                         #!Create new program file
#FOR(%Module)
  #CREATE(%Module & '.CLW')                      #!Make module files
#ENDFOR
#REDIRECT(%NewProgramFile)                       #!Redirect output to program file
#GENERATE(%Program)                              #!Generate main program header
#CLOSE(%NewProgramFile)                          #!Create new program file
#FOR(%Module)
  #REDIRECT(%Module & '.CLW')                    #!Redirect output to module file
  #GENERATE(%Module)                             #!Generate module header
  #FOR(%ModuleProcedure)                         #!For all procs in module
    #FIX(%Procedure,%ModuleProcedure)            #!Fix current procedure
    #GENERATE(%Procedure)                        #!Generate procedure code
  #ENDFOR                                        #!EndFor all procs in module
#ENDFOR
#!The following code demonstrates the LIFO files list:
#REDIRECT('F1.CLW')                              #!List contains:   F1
#REDIRECT('F2.CLW')                              #!List contains:   F1,  F2
#REDIRECT('F3.CLW')                              #!List contains:   F1,  F2,  F3
#REDIRECT( )                                     #!List contains:   F1,  F2
#REDIRECT( )                                     #!List contains:   F1
```

#REJECT (section invalid for use)

#REJECT

The **#REJECT** statement terminates #RESTRICT processing, indicating that the Template Code Section (#CODE, #CONTROL, #EXTENSION, #PROCEDURE, #PROGRAM, or #MODULE) is invalid.

The #RESTRICT structure contains Template language *statements* that evaluate the propriety of generating the section's source code. The #ACCEPT statement may be used to explicitly declare the section as appropriate. An implicit #ACCEPT also occurs if the #RESTRICT *statements* execute without encountering a #REJECT statement. The #REJECT statement must be used to specifically exclude the section from use. Both the #ACCEPT and #REJECT statements immediately terminate processing of the #RESTRICT code.

Example:

```
#CODE(ChangeControlSize,'Change control size')
  #WHERE(%EventHandling)
  #RESTRICT
    #CASE(%ControlType)
    #OF 'LIST'
    #OROF 'BUTTON'
      #REJECT
    #ELSE
      #ACCEPT
    #ENDCASE
  #ENDRESTRICT
  #PROMPT('Control to change',CONTROL),%MyField,REQ
  #PROMPT('New Width',@n04),%NewWidth
  #PROMPT('New Height',@n04),%NewHeight
%MyField{PROP:Width} = %NewWidth
%MyField{PROP:Height} = %NewHeight
```

See Also: #RESTRICT

#ACCEPT

#RELEASE (commit conditional source generation)

#RELEASE

The **#RELEASE** enables source generation in a #SUSPEND section. This allows empty unnecessary "boiler-plate" code to be easily removed from the generated source. The code in a #SUSPEND section is generated only when a #RELEASE statement is encountered.

#SUSPEND sections may be nested within each other to as many levels as necessary. A #RELEASE encountered in an inner nested section commits source generation for all the outer nested levels in which it is contained, also.

A #EMBED that contains source to generate performs an implied #RELEASE. Any generated source output also performs an implied #RELEASE. Therefore, an explicit #RELEASE statement is not always necessary. The #? statement defines an individual conditional source line that does not perform the implied #RELEASE.

Example:

```
ACCEPT
#SUSPEND                     #!Begin suspended generation
  #?CASE SELECTED()
  #FOR(%ScreenField)
    #SUSPEND
  #?OF %ScreenField
    #EMBED(%ScreenSetups,'Control selected code'),%ScreenField
    #!Implied #RELEASE from the #EMBED of both nested sections
    #RESUME
  #?END
#RESUME                      #!End suspended generation

#SUSPEND                     #!Begin suspended generation
  #?CASE EVENT()
  #SUSPEND
  #?OF EVENT:AlertKey
    #SUSPEND
    #?CASE KEYCODE()
      #FOR %HotKey
        #RELEASE             #!Explicit #RELEASE
    #?OF %HotKey
      #EMBED(%HotKeyProc,'Hot Key code'),%HotKey
      #ENDFOR
    #?END
    #RESUME
  #?END
#RESUME                      #!End suspended generation
END
```

```
    IF scField
        scField{Prop:Background} = scColor
#!
#! test for style
#SUSPEND
        #?scField{Prop:FontStyle}  = scStyle
#IF(%SelectedStyle = %True)
    #RELEASE
#ENDIF
#RESUME
#!
#! test for caret
#SUSPEND
        #?HIDE(LocSelectedCaret)
#IF(%SelectedCaret = %True)
    #RELEASE
#ENDIF
#RESUME
#!
    END
```

See Also: #SUSPEND

#RESUME

#?

#REMOVE (delete a source file)

#REMOVE(*file* **)**

#REMOVE	Deletes a source output file.
file	A string constant, template symbol, or expression containing a DOS file specification. This may be a fully qualified DOS pathname.

The **#REMOVE** statement deletes the specified source output *file*. If the *file* does not exist, #REMOVE is ignored and source generation continues.

Example:

```
#FOR(%Module)
   #SET(%TempModuleFile,(%Module & '.$$$'))           #!Set temp module file
   #CREATE(%TempModuleFile)                           #!Create temp module file
   #FOR(%ModuleProcedure)                             #!For all procs in module
      #FIX(%Procedure,%ModuleProcedure)               #!Fix current procedure
      #GENERATE(%Procedure)                           #!Generate procedure code
   #ENDFOR                                            #!EndFor all procs in module
   #SET(%ModuleFile,(%Module & '.CLW'))               #!Set to current module file
   #CREATE(%ModuleFile)                               #!Create module file
   #GENERATE(%Module)                                 #!Generate module header
   #APPEND(%TempModuleFile)                           #!Add generated procedures
   #REMOVE(%TempModuleFile)                           #!Delete the temporary file
#ENDFOR
```

#REPLACE (conditionally replace source file)

#REPLACE(*oldfile, newfile* **)**

#REPLACE	Performs "intelligent" file replacement.
oldfile	A string constant, template symbol, or expression containing a DOS file specification. This may be a fully qualified DOS pathname.
newfile	A string constant, template symbol, or expression containing a DOS file specification. This may be a fully qualified DOS pathname.

The **#REPLACE** statement performs a binary comparison between the contents of the *oldfile* and *newfile*. If the contents of the *oldfile* are different from the contents of the *newfile* (or the *oldfile* does not exist), then the *oldfile* is deleted and the *newfile* is renamed to the *oldfile*. If the two files are identical, then no action is taken. If the *newfile* does not exist, #REPLACE is ignored and source generation continues.

Example:

```
#FOR(%Module)
   #SET(%TempModuleFile,(%Module & '.$$$'))          #!Set temp module file
   #CREATE(%TempModuleFile)                          #!Create temp module file
   #GENERATE(%Module)                                #!Generate module header
   #FOR(%ModuleProcedure)                            #!For all procs in module
      #FIX(%Procedure,%ModuleProcedure)              #!Fix current procedure
      #GENERATE(%Procedure)                          #!Generate procedure code
   #ENDFOR                                           #!EndFor all procs in module
   #SET(%ModuleFile,(%Module & '.CLW'))              #!Set to existing module file
   #REPLACE(%ModuleFile,%TempModuleFile)             #!Replace old with new (if changed
#ENDFOR
```

#REPORTS (default report structures)

#REPORTS

structures

#ENDREPORTS

#REPORTS	Begins a default report data structure section.
structures	Default REPORT structures.
#ENDREPORTS	Terminates the default report section.

The **#REPORTS** structure contains default REPORT data structures for a procedure Template. The default report *structures* provide a starting point for the procedure's report design.

The #REPORTS section may contain multiple *structures* which may be chosen as the starting point for the procedure's report design. If there is more than one report structure to choose from, the Application Generator displays a list of those *structures* available the first time the procedure's report is editted. The names of the windows which appear in the Application Generator's list comes from a preceding comment beginning with two exclamations and a right angle bracket (!!>).

If the procedure template contains a #DEFAULT procedure, there is no need for #REPORT, since the default report is already in the #DEFAULT. Therefore, the list does not appear when the report is first editted.

Example:

```
#REPORTS
!!> Report
Label  REPORT,AT(1000,2500,6000,6000),THOUS
         HEADER,AT(1000,1000,6000,1000)
         END
Detail   DETAIL
         END
         FOOTER,AT(1000,10000,6000,1000)
         END
         FORM,AT(1000,1000,6000,9000)
         END
       END
#ENDREPORTS
```

#RESTRICT (define section use constraints)

#RESTRICT [, **WHERE(** *expression* **)**]

statements

#ENDRESTRICT

#RESTRICT	Specifies conditions where a Template Code Section (#CODE, #CONTROL, #EXTENSION, #PROCEDURE, #PROGRAM, or #MODULE) can be used.
WHERE	The #RESTRICT *statements* are excuted only when the *expression* is true.
expression	A logical expression to limit execution of the #RESTRICT *statements*.
statements	Template language code to #ACCEPT or #REJECT use of the section which contains the #RESTRICT structure.
#ENDRESTRICT	Terminates the #RESTRICT structure.

The **#RESTRICT** structure provides a mechanism to limit the availability of a Template Code Section (#CODE, #CONTROL, #EXTENSION, #PROCEDURE, #PROGRAM, or #MODULE) at application design time to only those points where the generated code would be appropriate. Any WHERE clause on the Template Code Section is evaluated first, before #RESTRICT.

The #ACCEPT statement may be used to explicitly declare the section as appropriate for use. An implicit #ACCEPT also occurs if the #RESTRICT *statements* execute without encountering a #REJECT statement. The #REJECT statement must be used to specifically exclude the section from use. Both the #ACCEPT and #REJECT statements immediately terminate processing of the #RESTRICT code.

Example:

```
#CODE(ChangeControlSize,'Change control size')
  #RESTRICT
    #CASE(%ControlType)
    #OF('LIST')
    #OROF('BUTTON')
      #REJECT
    #ELSE
      #ACCEPT
    #ENDCASE
  #ENDRESTRICT
  #PROMPT('Control to change',CONTROL),%MyField,REQ
  #PROMPT('New Width',@n04),%NewWidth
  #PROMPT('New Height',@n04),%NewHeight
%MyField{PROP:Width} = %NewWidth
%MyField{PROP:Height} = %NewHeight
```

See Also: #ACCEPT

#REJECT

#RESUME (delimit conditional source)

#RESUME

The **#RESUME** statement marks the end of a section of source that is generated only if a #RELEASE statement is encountered. This allows empty unnecessary "boiler-plate" code to be easily removed from the generated source. The beginning of the section must be delimited by a matching #SUSPEND statement.

These #SUSPEND sections may be nested within each other to as many levels as necessary. A #RELEASE encountered in an inner nested section commits source generation for all the outer nested levels in which it is contained, also.

A #EMBED that contains source to generate performs an implied #RELEASE. Any generated source output also performs an implied #RELEASE. Therefore, an explicit #RELEASE statement is not always necessary. The #? statement defines an individual conditional source line that does not perform the implied #RELEASE.

When a #RESUME is executed without the output to the file being released, any conditional lines of code are un-done back to the matching #SUSPEND.

Example:

```
ACCEPT
#SUSPEND                        #!Begin suspended generation
  #?CASE SELECTED()
  #FOR(%ScreenField)
    #SUSPEND
  #?OF %ScreenField
    #EMBED(%ScreenSetups,'Control selected code'),%ScreenField
    #!Implied #RELEASE from the #EMBED of both nested sections
    #RESUME
  #?END
#RESUME                         #!End suspended generation
#SUSPEND                        #!Begin suspended generation
  #?CASE EVENT()
  #SUSPEND
  #?OF EVENT:AlertKey
    #SUSPEND
    #?CASE KEYCODE()
      #FOR %HotKey
        #RELEASE #!Explicit #RELEASE
    #?OF %HotKey
      #EMBED(%HotKeyProc,'Hot Key code'),%HotKey
      #ENDFOR
    #?END
    #RESUME
  #?END
#RESUME                         #!End suspended generation
END
```

See Also: #SUSPEND

#RELEASE

#?

#RETURN (return from #GROUP)

#RETURN([*returnvalue*] **)**

#RETURN	Immediately returns from the #GROUP.
returnvalue	An expression containing the value to return to the calling statement. If omitted, an empty string ('') is returned.

The **#RETURN** statement immediately returns control to the statement following the #INSERT, #CALL, or #INVOKE that called the #GROUP, or returns the *returnvalue* to the CALL or INVOKE built-in procedures. If the #INSERT, #CALL, or #INVOKE that called the #GROUP are set to receive the *returnvalue*, the value is placed in the symbol named in the #INSERT, #CALL, or #INVOKE statement.

#RETURN is only valid in a #GROUP section.

Example:

```
#GROUP(%ProcessListGroup,%PassedControl)
  #FIX(%ScreenField,%PassedControl)
  #IF (%ScreenFieldType <> 'LIST')
    #UNFIX(%ScreenField)
    #RETURN
  #ENDIF
```

See Also: #GROUP

#INSERT

#CALL

CALL

#INVOKE

INVOKE

#RUN (execute program)

#RUN(*program* **)** [, **WAIT**]

#RUN	Launches the named *program*.
program	A string constant containing the name of the program to execute, including any command line parameters.
WAIT	Specifies waiting for the *program* to complete its execution before proceeding to the next template language statement..

The **#RUN** statement launches the specified *program* by using the Clarion language RUN statement. Template execution continues concurrently with the execution of the *program*, unless the WAIT attribute is specified.

Example:

```
#RUN('MyExe.EXE')                             #!Launch MyExe.EXE
```

#RUNDLL (execute DLL procedure)

#RUNDLL(*library, procedure* [, *parameter*] **)** [, **| RETAIN |**] [, **WIN32**]
| RELEASE |

#RUNDLL	Executes a procedure from any Windows-standard Dynamic Link Library (.DLL).
library	A string constant containing the name of the .DLL file, including the extension.
procedure	A string constant containing the name of the procedure in the DLL file to execute.
parameter	A string constant or expression containing a single parameter to pass to the *procedure*.
RETAIN	Specifies that the *library* stays in memory after the *procedure* has completed execution.
RELEASE	Specifies that the *library* is unloaded from memory after the *procedure* has completed execution.
WIN32	Specifies that the *library* is a 32-bit DLL.

The **#RUNDLL** statement executes the specified *procedure* from the specified *library*. The *library* .DLL is dynamically loaded before the *procedure* executes and unloaded after, unless the RETAIN attribute is specified.

If the RETAIN attribute is specified, another #RUNDLL statement with the RELEASE attribute and naming the same *library* must follow so that the .DLL is unloaded from memory. Failure to unload the .DLL can result in memory leaks. The RETAIN and RELEASE attributes can be nested as long as the #RUNDLL statements match up.

If a *parameter* is named, then the *procedure* must accept a single *CSTRING parameter (only). If no *parameter* is named, then the *procedure* must not accept any parameters. In either case, the *procedure* cannot return a value. If the *parameter* named is a single-valued user-defined symbol, then any changes to the passed parameter's value in the called *procedure* are reflected in that *parameter* symbol.

Example:

```
!A .DLL 'Services.DLL' contains these two procedures:
SoundBeep PROCEDURE
 CODE
 BEEP

AskForName PROCEDURE(*CSTRING Name)
Window  WINDOW('Enter Name),AT(,,260,100),SYSTEM,GRAY,AUTO
         ENTRY(@s20),AT(4,4),USE(Name)
         BUTTON('OK'),AT(200,4),USE(?Ok),STD(STD:Close)
        END
 CODE
 OPEN(Window)
 ACCEPT
 END

! The Template code that calls the procedures in Services.DLL:
#DECLARE (%UserName)
#SET (%UserName, 'Unknown')
#RUNDLL ('Services.DLL', 'SoundBeep'),RETAIN       !Use RETAIN to improve performance
#RUNDLL ('Services.DLL', 'AskForName', %UserName)
#RUNDLL ('Services.DLL', 'SoundBeep'),RELEASE      !RELEASE matches the RETAIN
```

#SECTION (define code section)

#SECTION

statements

#ENDSECTION

#SECTION	Marks the beginning of a section of generated code.
statements	Any template or target language statements.
#ENDSECTION	Terminates the #SECTION.

The **#SECTION** structure defines a contiguous section of generated code so that the position of embedded source code is correctly patched up for error positions in the resulting generated code. #APPEND with the SECTION attribute perfroms the patch.

Example:

```
#CREATE(%Temp)                                  #!Create temp module file
#SECTION
#EMBED(%MyEmbed,@s30)
#ENDSECTION
#CLOSE
#CREATE('target.clw')
 CODE
#APPEND(%Temp),SECTION                          #!Correctly patch position of
#EMBED
#CLOSE
```

See Also: #APPEND

Named #SECTIONS

Named #SECTIONS

A new parameter has been added to the #SECTION statement.

Example:

```
#SECTION('First Section')
...
#ENDSECTION

#SECTION('Another Section')
...
#ENDSECTION
...
```

This parameter can then be referenced by the SECTION attribute of the #APPEND statement as follows:

```
#APPEND('FileName'),SECTION('Another Section')
#APPEND('FileName'),SECTION('First Section')
```

The first #APPEND statement with the SECTION attribute uses the last defined section in source order. The next #APPEND statement is used for the previous section, etc

A named #SECTION can be used with the #APPEND statement **only once**.

If the SECTION attribute of the #APPEND statement has a parameter defined, for example:

```
#APPEND('somefile'),SECTION('somesection')
```

the template source generator will append the #SECTION defined with the name 'somesection' and then removes it from the list of sections.

If the SECTION attribute has no parameter

```
#APPEND('somefile'),SECTION
```

the generator uses the most recently declared #SECTION not removed yet from the top of sections stack and then pops it (but does not destroy).

In summary, to avoid any logic problems, all #APPEND statements ***should not*** mix usage of named and unnamed sections.

#SELECT (fix a multi-value symbol)

#SELECT(*symbol, instance* **)**

#SELECT	Fixes a multi-valued symbol to a particular *instance* number.
symbol	A multi-valued symbol.
instance	An expression containing the number of the instance to which to fix.

The **#SELECT** statement fiXEs the current value of the multi-valued *symbol* to a specific *instance*. The result of #SELECT is exactly the same as #FIX. Each *instance* in the multi-valued *symbol* is numbered starting with one (1).

The *instance* must contain a valid instance number of one of the *symbol's* multiple values. If the *instance* is not valid, the *symbol* is cleared and contains no value when referenced. The INSTANCE built-in template procedure can return the instance number.

Unless #ADD has been used to add a new value and fix to that instance, #FIX or #SELECT must be used to set the value in a *symbol* before it contains any value for Template processing outside of a #FOR loop.

Example:

```
#SELECT(%File,1)     #!Fix to first %File instance
```

#SERVICE (internal use only)

#SERVICE

The **#SERVICE** statement is for internal use only. It is not documented and is listed here simply because it is visible in the shipping templates.

#SET (assign value to a user-defined symbol)

#SET(*symbol*,*value* **)**

#SET	Assigns a value to a single-valued user-defined symbol.
symbol	A single-valued user-defined symbol. This must have been previously declared with the #DECLARE statement.
value	A built-in or user-defined symbol, string constant, or an expression.

The **#SET** statement assigns the *value* to the *symbol*. If the *value* parameter contains an expression, you may perform mathematics during source code generation. The expression may use any of the arithmetic, Boolean, and logical operators documented in the *Language Reference*. If the modulus division operator (%) is used in the expression, it must be followed by at least one blank space (to explicitly differentiate it from the Template symbols). Logical expressions always evaluate to 1 (True) or 0 (False). Clarion language procedure calls (those supported in EVALUATE()) and built-in template procedures are allowed.

Example:

```
#SET(%NetworkApp,'Network')
#SET(%MySymbol,%Primary)
#FOR(%File)
  #SET(%FilesCounter,%FilesCounter + 1)
%FileStructure
#ENDFOR
```

See Also: #DECLARE

#SHEET (declare a group of #TAB controls)

#SHEET [, **HSCROLL**] [, **ADJUST**]

tabs

#ENDSHEET

#SHEET	Declares a group of #TAB controls.
HSCROLL	Specifies scrolling #TAB controls.
ADJUST	Specifies automatic prompt positioning if the #TAB controls create more than one row of tabs.
tabs	Multiple #TAB control declarations.
#ENDSHEET	Terminates the group box of *prompts*.

#SHEET declares a group of #TAB controls that offer the user multiple "pages" of prompts for the window. The multiple #TAB controls in the SHEET structure define the "pages" displayed to the user.

Example:

```
#UTILITY(ApplicationWizard,'Create a New Database Application'),WIZARD
#!
#SHEET
  #TAB('Application Wizard')
    #IMAGE('CMPAPP.BMP')
    #DISPLAY('This wizard will create a new Application.'),AT(90,8,235,24)
    #DISPLAY('To begin creating your new Application, click Next.'),AT(90)
  #ENDTAB
  #TAB('Application Wizard - File Usage'),FINISH(1)
    #IMAGE('WINAPP.BMP')
    #DISPLAY('You can gen procs for all files, or select them'),AT(90,8,235,24)
    #PROMPT('Use all files in DCT',CHECK),%GenAllFiles,AT(90,,180),DEFAULT(1)
  #ENDTAB
  #TAB('Select Files to Use'),WHERE(NOT %GenAllFiles),FINISH(1)
    #IMAGE('WINAPP.BMP')
    #PROMPT('File Select',FROM(%File)),%FileSelect,INLINE,SELECTION('File Select')
  #ENDTAB
  #TAB('Application Wizard - Finally...'),FINISH(1)
    #IMAGE('WINAPP.BMP')
    #DISPLAY('Old procs can be overwritten or new procs suppressed')
    #PROMPT('Overwrite existing procs',CHECK),%OverwriteAll,AT(90,,235),DEFAULT(0)
    #IMAGE('<255,1,4,127>'),AT(90,55)
    #DISPLAY('Your First Procedure is always overwritten!'),AT(125,54,200,20)
  #ENDTAB
#ENDSHEET
```

#SUSPEND (begin conditional source)

#SUSPEND

The **#SUSPEND** statement marks the start of a section of source that is generated only if a non-conditional source line or a #RELEASE statement is encountered. (i.e., a line of code without "#?" or a #RELEASE must be encountered for the code between #SUSPEND and #RESUME to be generated). This allows empty unnecessary "boiler-plate" code to be easily removed from the generated source. The end of the section must be delimited by a matching #RESUME statement.

These #SUSPEND sections may be nested within each other to as many levels as necessary. A #RELEASE encountered in an inner nested section commits source generation for all the outer nested levels in which it is contained, also.

A #EMBED that contains source to generate performs an implied #RELEASE. Any generated source output also performs an implied #RELEASE. Therefore, an explicit #RELEASE statement is not always necessary. The #? statement defines an individual conditional source line that does not perform the implied #RELEASE.

Example:

```
ACCEPT
#SUSPEND                        #!Begin suspended generation
  #?CASE SELECTED()
  #FOR(%ScreenField)
    #SUSPEND
  #?OF %ScreenField
    #EMBED(%ScreenSetups,'Control selected code'),%ScreenField
    #!Implied #RELEASE from the #EMBED of both nested sections
    #RESUME
  END
#RESUME                         #!End suspended generation
#SUSPEND                        #!Begin suspended generation
  #?CASE EVENT()
  #SUSPEND
  #?OF EVENT:AlertKey
    #SUSPEND
    #?CASE KEYCODE()
      #FOR %HotKey
        #RELEASE                #!Explicit #RELEASE
    #?OF %HotKey
      #EMBED(%HotKeyProc,'Hot Key code'),%HotKey
      #ENDFOR
    #?END
    #RESUME
  #?END
#RESUME                         #!End suspended generation
END
```

See Also: #RELEASE, #RESUME, #?

#SYSTEM (template registration and load)

#SYSTEM

The **#SYSTEM** statement marks the beginning of a section of Template code which executes when the Template set is registered or loaded from the template registry. The section is terminated by the next Template Code Section (#PROGRAM, #MODULE, #PROCEDURE, #CONTROL, #CODE, #EXTENSION, #UTILITY, or #GROUP) statement. Any #PROMPT statements in a #SYSTEM section appear on the Application Options dialog, once the Template Registry has been loaded (opened an .APP file).

Example:

```
#SYSTEM
#TAB('Wizards')
#PROMPT('Use Toolbar mode for Wizards',CHECK),%UseToolBarFor Wizard
#ENDTAB
#DECLARE(%GlobalSystemSymbol)
```

#TAB (declare a page of a #SHEET control)

#TAB(*text* **)** [,**FINISH(** *expression* **)**] [,**WHERE(** *expression* **)**]

prompts

#ENDTAB

#TAB	Declares a group of *prompts* that constitute one of the multiple "pages" within a #SHEET structure.
text	A string constant containing the text to display on the tab, or as the title of the window, if the WIZARD attribute is present on the #UTILITY.
FINISH	Specifies the "Finish" button is present. Valid only in a #UTILITY with the WIZARD attribute.
expression	An expression that specifies whether the "Finish" button is enabled or disabled. If the expression is true, the button is enabled, if false, the button is disabled.
WHERE	Specifies the #BOXED is visible only for those instances where the *expression* is true.
expression	An expression that specifies the condition for use.
prompts	One or more #PROMPT statements. This may also contain #DISPLAY, #VALIDATE, #ENABLE, and #BUTTON statements.
#**ENDTAB**	Terminates the page of *prompts*.

The **#TAB** structure declares a group of *prompts* that constitute one of the multiple "pages" of controls contained within a #SHEET structure. The multiple #TAB controls in the #SHEET structure define the "pages" displayed to the user.

Example:

```
#UTILITY(ApplicationWizard,'Create a New Database Application'),WIZARD
#!
#SHEET
  #TAB('Application Wizard'),FINISH(0)             #!Finish button dim
    #IMAGE('CMPAPP.BMP')
    #DISPLAY('This wizard will create a new Application.'),AT(90,8,235,24)
    #DISPLAY('To begin creating your new Application, click Next.'),AT(90)
  #ENDTAB
  #TAB('Application Wizard - File Usage'),FINISH(1)       #!Finish button active
    #IMAGE('WINAPP.BMP')
    #DISPLAY('You can gen procs for all files, or select them'),AT(90,8,235,24)
    #PROMPT('Use all files in DCT',CHECK),%GenAllFiles,AT(90,,180),DEFAULT(1)
  #ENDTAB
  #TAB('Select Files to Use'),WHERE(NOT %GenAllFiles),FINISH(1)
    #IMAGE('WINAPP.BMP')
    #PROMPT('File Select',FROM(%File)),%FileSelect,INLINE,SELECTION('File Select')
  #ENDTAB
  #TAB('Application Wizard - Finally...'),FINISH(1)
    #IMAGE('WINAPP.BMP')
    #DISPLAY('Old procs can be overwritten or new procs suppressed')
    #PROMPT('Overwrite existing procs',CHECK),%OverwriteAll,AT(90,,235),DEFAULT(0)
    #IMAGE('<255,1,4,127>'),AT(90,55)
    #DISPLAY('Your First Procedure is always overwritten!'),AT(125,54,200,20)
  #ENDTAB
#ENDSHEET
```

#TEMPLATE (begin template set)

#TEMPLATE(*name, description* **)** [, **PRIVATE**] [, **FAMILY(** *sets* **)**]

#TEMPLATE	Begins the Template set.
name	The name of the Template set which uniquely identifies it for the Template Registry and Template Language statements. This must be a valid Clarion label.
description	A string constant describing the Template set for the Template Registry and Application Generator.
PRIVATE	Prevents re-generation from the Template Registry.
FAMILY	Names the other Template *sets* in which the #TEMPLATE is valid for use.
sets	A string constant containing a comma delimited list of the *names* of other Template sets in which the #TEMPLATE is valid for use.

The **#TEMPLATE** statement marks the beginning of a Template set. This should be the first non-comment statement in the Template file. The Template Registry allows multiple Template sets to be registered for the Application Generator.

Each Template Code Section (#APPLICATION, #PROGRAM, #MODULE, #PROCEDURE, #CONTROL, #CODE, #EXTENSION, and #GROUP) within a Template is uniquely identified by its #TEMPLATE statement's *name* and the name of the section. This allows different Template sets to contain Template Code Sections with names that duplicate those in other Template sets without ambiguity, and allows the programmer to concurrently use Template sets from multiple sources to generate applications.

Example:

```
#TEMPLATE(SampleTemplate,'This is a sample Template'),PRIVATE,FAMILY('ABC')
#INCLUDE('FileTwo.TPW')
#INCLUDE('FileThree.TPW')
```

#TYPEMAP (map EQUATE to data type)

#TYPEMAP (*equate data type*, *32-bit representation* [,*16-bit representation*] **)**

#TYPEMAP	Maps an EQUATE data type to a true data type based on the target operating system.
equate data type	A string constant containing the label of an equate used to represent a data type.
32-bit data type	A string constant containing the label of a data type that represents the equated data type in a 32-bit program.
16-bit data type	A string constant containing the label of a data type that represents the equated data type in a 16-bit program.(deprecated)

The **#TYPEMAP** statement allows the template to map a data type EQUATE to a true data type based on the target operating system of the application. This allows the compiler to resolve these EQUATES to their respective data types.

Example:

```
#TYPEMAP('SIGNED','LONG')
#TYPEMAP('SIGNED','LONG','SHORT')
```

#UNFIX (unfix a multi-value symbol)

#UNFIX(*symbol* **)**

#UNFIX	UnfiXEs a multi-valued symbol.
symbol	A multi-valued symbol.

The **#UNFIX** statement unfiXEs the current value of the multi-valued *symbol*. If the unfiXEd *symbol* is referenced outside a #FOR loop structure, it has no value and you cannot reference any other symbols dependent upon the multi-valued *symbol*.

Example:

```
#SET(%OneFile,'HEADER')         #! Put values into two User-defined symbols
#SET(%TwoFile,'DETAIL')
#FIX(%File,%OneFile)            #! %File refers to 'HEADER'
#FOR(%File)                     #! %File iteratively refers to all file names
  #FIX(%File,%TwoFile)          #! %File refers to 'DETAIL'
#ENDFOR
                                #! %File refers to 'HEADER' again
#UNFIX(%File)                   #! %File refers to no spcific value
```

#UTILITY (utility execution section)

#UTILITY(*name, description* **)** [, **HLP(** *helpid* **)**] [, **WIZARD(** *procedure* **)**]

#UTILITY	Begins a utility generation control.
name	The name of the #UTILITY which identifies it for the Template Registry. This must be a valid Clarion label.
description	A string constant describing the utility section.
HLP	Specifies on-line help is available.
helpid	A string constant containing the identifier to access the Help system. This may be either a Help keyword or "context string."
WIZARD	Specifies the #UTILITY is used as a Wizard to generate a procedure or a complete application.
procedure	A string constant containing the fully qualified name of the #PROCEDURE for which it is a WIZARD.

The **#UTILITY** statement marks the beginning of a utility execution control section. The section is terminated by the next Template Code Section (#PROGRAM, #MODULE, #PROCEDURE, #CONTROL, #CODE, #EXTENSION, #UTILITY, or #GROUP) statement. The Template statements contained in this section control the utility execution process. Multiple #UTILITY sections are allowed in a single Template set.

The #UTILITY section is very similar to the #APPLICATION section, in that it allows you to produce output from the application. The purpose of #UTILITY is to provide extensible supplemental utilities for such things as program documentation, or a tree diagram of procedure calls. The list of registered utilities appears in the Utilities menu in the Clarion for Windows environment. #UTILITY with the WIZARD attribute specifies it contains a #SHEET with #TABs that display one tab at a time, guiding the user through the prompts.

Example:

```
#UTILITY(ProcCallTree, 'Output procedure call tree')
  #CREATE(%Application & '.TRE')
Procedure Call Tree: for %Application
#INSERT(%DisplayTree, %FirstProcedure, '', '  ')
#CLOSE
#!***********************************************************
#GROUP(%DisplayTree, %ThisProc, %Level, %NextIndent)
  #FIX(%Procedure, %ThisProc)
%Level+-%ThisProc (%ProcedureTemplate)
  #FOR(%ProcedureCalled)
    #IF(INSTANCE(%ProcedureCalled) = ITEMS(%ProcedureCalled))
#INSERT(%DisplayTree, %ProcedureCalled, %Level & %NextIndent, '  ')
    #ELSE
#INSERT(%DisplayTree, %ProcedureCalled, %Level & %NextIndent, '| ')
    #ENDIF
  #ENDFOR
```

#VALIDATE (validate prompt input)

#VALIDATE(*expression,message* **)**

#VALIDATE	Validates the data entered into the immediately preceding #PROMPT field.
expression	The expression to use to validate the entered data.
message	A string constant containing the error message to display if the data is invalid.

The **#VALIDATE** statement validates the data entered into the #PROMPT field immediately preceding the #VALIDATE. The *expression* is evaluated when the OK button is pressed on the Procedure Properties window. If the *expression* is false, the *message* is displayed to the programmer in a message box, and control is given to the #PROMPT field that immediately precedes the #VALIDATE. There may be multiple #VALIDATE statements following a #PROMPT to validate the entry.

Example:

```
#PROMPT('Input Value, Even numbers from 100-200',@N3),%Value
#VALIDATE((%Value > 100) AND (%Value < 200),'Value must be between 100 and 200')
#VALIDATE((%Value % 2 = 0),'Value must be an even number')
#PROMPT('Screen Field',CONTROL),%SomeField
#VALIDATE(%ScreenFieldType = 'LIST','Must select a list box')

#PROMPT('Select Change Date/Time Group:',FIELD(%Primary)), %ChangeDateGroup,REQ
#PREPARE
#FIX(%File,%Primary)
#FIX(%Field,%ChangeDateGroup)
#ENDPREPARE
#VALIDATE(%FieldType = 'GROUP','Change Date Group must be of type GROUP!')
```

See Also: #PROMPT

#WHERE (define #CODE embed point availability)

#WHERE(*embeds* **)**

#WHERE	Limits the availability of a #CODE to only those specific embedded source code points where the generated code would be appropriate.
embeds	A comma-delimited list of #EMBED *identifiers* that specifies the embedded source code points that may use the #CODE to generate source code.

The **#WHERE** statement limits the availability of a #CODE to only those #EMBED embedded source code points where the generated code would be appropriate. A single #CODE may contain multiple #WHERE statements to explicitly define all the valid #EMBED embedded source code points. All the #WHERE statements in a #CODE are evaluated to determine which embedded source code points have been specifically enabled.

The *embeds* list must contain individual #EMBED *identifiers* delimited by commas. It may also contain ranges of embed points in the form *FirstIdentifier..LastIdentifier*, also delimited by commas. The *embeds* list may contain both types in a "mix and match" manner to define all suitable embedded source code points.

Example:

```
#CODE(ChangeProperty,'Change control property')
  #WHERE(%AfterWindowOpening..%CustomRoutines)
  #!Appropriate everywhere after window open
  #PROMPT('Control to change',CONTROL),%MyField,REQ
  #PROMPT('Property to change',@S20),%MyProperty,REQ
  #PROMPT('New Value',@S20),%MyValue,REQ
%MyField{%MyProperty} = '%'MyValue
```

See Also:	#EMBED
	#CODE
	#RESTRICT

#WINDOWS (default window structures)

#WINDOWS

structures

#ENDWINDOWS

#WINDOWS	Begins a default window data structure section.
structures	Default APPLICATION or WINDOW structures.
#ENDWINDOWS	Terminates the default window section.

The **#WINDOWS** structure contains default APPLICATION or WINDOW data structures for a procedure Template. The default window *structures* provide a starting point for the procedure's window design.

The #WINDOWS section may contain multiple *structures* which may be chosen as the starting point for the procedure's window design. If there is more than one window structure to choose from, the Application Generator displays a list of those *structures* available the first time the procedure's window is editted. The names of the windows which appear in the Application Generator's list comes from a preceding comment beginning with two exclamations and a right angle bracket (!!>).

If the procedure template contains a #DEFAULT procedure, there is no need for #WINDOWS, since the default window is already in the #DEFAULT. Therefore, the list does not appear when the window is first edited.

Example:

```
#WINDOWS
!!> Window
Label  WINDOW('Caption'),AT(0,0,100,100)
       END
!!> Window with OK & Cancel
Label  WINDOW('Caption'),AT(0,1,185,92)
         BUTTON('OK'),AT(144,10,35,14),DEFAULT,USE(?Ok)
         BUTTON('Cancel'),AT(144,28,36,14),USE(?Cancel)
       END
#ENDWINDOWS
```

#WITH (associate prompts with a symbol instance)

#WITH(*symbol* , *value* **)**

prompts

#ENDWITH

#WITH	Allows a set of prompts to be associated with a single instance of a *symbol* (as specified by the *value* expression).
symbol	A previously declared unique multi-valued template symbol.
value	A string expression containing a unique value to add to the *symbol*.
prompts	One or more #PROMPT statements whose values become dependent on the value of the *symbol*. This may also contain #DISPLAY, #VALIDATE, #ENABLE, and #BUTTON statements.
#ENDWITH	Terminates the group of *prompts* associated with the #WITH *symbol*.

The **#WITH** statement adds a new instance to the *symbol* (as specified by the *value* expression) and makes the values of all the *prompts* dependent upon that instance. #WITH allows the same set of prompts, with the same prompt symbols, to be used more than once on the same screen.

Example:

```
#WITH(%ClassItem,'Default')
 #INSERT(%ClassPrompts)
#ENDWITH
```

See Also:	#PROMPT
	#VALIDATE
	#ENABLE
	#DISPLAY
	#BUTTON

4 - Template Symbols

Symbol Overview

The Clarion Template Language uses symbols which act as variables do in a programming language -- they contain information that can be used as-is or may be used in expressions. These symbols may come from the built-in symbol set, or can be defined by the template author. Both types may be single-valued or multi-valued.

The built-in symbols that are available to the Template writer contain information from both the Dictionary and the Application about how the programmer has designed the application. The template-defined symbols contain information provided by the programmer from prompts on the Application Generator's properties windows, or may only be for internal use.

All template symbols expand during source generation to place the value they contain in the generated source code (if included in template code that generates source).

See Also:

Expansion Symbols

Symbol Hierarchy Overview

Expansion Symbols

There are several special symbol forms that expand to allow formatting and special characters to generate into the source. These may be combined with each other to produce complex effects.

%%	Expands to a single percent (%) sign. This allows the Application Genetrator to generate the modulus operator without confusion with any symbol.
%#	Expands to a single pound (#) sign. This allows the Application genetrator to generate an implicit LONG variable without confusion with any Template Language statement.
%@*picture*@*symbol*	Formats the *symbol* with the specified *picture* when source generates. For example, %@D1@MyDate expands the %MyDate symbol, formatted for the @D1 *picture*.
%[*number*]*symbol*	Expands the *symbol* to fill at least the *number* of spaces specified. This allows proper comment and data type alignment in the generated source.
%|	Expands the next generated source onto the same line as the last. This is the Template line continuation character.
%'*symbol*	Expands the *symbol's* string data, "doubling up" single quotes ('), and all un-paired left angle brackets (<) and left curly braces ({) to prevent compiler errors.
%(*expression*)	Expands the *expression* into the generated source.

Example:

```
%(ALL(' ', %indent))%[20]Field %@d3@Date  #!Generate an indent, expand %Field
                                          #!to occupy at least 20 spaces, then
                                          #! generate the date in mmm dd, yyyy format

%[30]Null  #!Generate 30 spaces

#! %MySymbol contains: Gavin's Holiday
StringVar = '%'MySymbol'                  #!Expands as a valid Clarion string constant
                                          #! to 'Gavin''s Holiday"
```

Symbol Hierarchy Overview

The Built-in Symbols all form a hierarchy of dependencies. This hierarchy starts with %Application, upon which all the other built-in symbols are dependent. The following tree diagram does not show all the dependent symbols, but does graphically represent the hierarchy of symbols. Most of these are multi-valued symbols.

- **%Application**
  - **%DictionaryFile**
    - **%File**
      - **%Field**
      - **%Key**
      - **%Relation**
  - **%Program**
  - **%GlobalData**
  - **%Module**
    - **%ModuleProcedure**
    - **%MapItem**
    - **%ModuleData**
  - **%Procedure**
    - **%Report**
      - **%ReportControl**
        - **%ReportControlField**
    - **%Window**
      - **%WindowEvent**
      - **%Control**
        - **%ControlEvent**
    - **%ProcedureCalled**
    - **%LocalData**
    - **%ActiveTemplate**
      - **%ActiveTemplateInstance**
    - **%Formula**
      - **%FormulaExpression**

These symbols (and all the symbols not listed here that are dependent upon these) contain all the information about the applciation that is available in the data dictionary (.DCT) and application (.APP) files. They enable you to write a template to generate any type of code you require.

Built-in Symbols

Symbols Dependent on %Application

%Application	The name of the .APP file. The hierarchy of built-in symbols starts with %Application.
%ApplicationDebug	Contains 1 if the application has debug enabled.
%ApplicationLocalLibrary	Contains 1 if the application is linking in the Clarion runtime library.
%ApplicationTemplate	The name of all global extension templates used in the application. Multi-valued.
%ApplicationTemplateInstance	The instance numbers of all global extension templates used in the application. Multi-valued. Dependent on %ApplicationTemplate.
%ApplicationTemplateInstanceDescription	The description of global extension templates used in the application. Dependent on %ApplicationTemplate.
%ApplicationTemplateParentInstance	The instance number of the control template's parent global extension template. This is the global extension template that it is "attached" to. Dependent on %ApplicationTemplateInstance.
%ApplicationTemplatePrimaryInstance	The instance number of the control template's primary global extension template. This is the global extension template in a succession of multiple related global extension templates. Dependent on %ApplicationTemplateInstance.
%ProjectTarget	Contains the name of the file being produced.
%Target32	Contains 1 if the application is producing a 32-bit program.
%DictionaryChanged	Contains 1 if the .DCT file has changed since the last source generation.
%RegistryChanged	Contains 1 if the .REGISTRY.TRF file has changed since the last source generation.
%ProgramDateCreated	The program creation date (a Clarion standard date).
%ProgramDateChanged	The date the program was last changed (a Clarion standard date).

%ProgramTimeCreated	The program creation time (Clarion standard time).
%ProgramTimeChanged	The time the program was last changed (a Clarion standard time).
%FirstProcedure	The label of the application's first procedure.
%HelpFile	The name of the application's help file.
%ProgramExtension	Contains EXE, DLL, or LIB.
%DictionaryFile	The name of the .DCT filefor the application.
%DictionaryQuickOptions	A comma-delimited string containing the choices the user made on the Options tab for the dictionary properties.
%DictionaryUserOptions	A string containing the entries the user made in the User Options text box on the Options tab for the dictionary properties.
%DictionaryToolOptions	A string containing the entries third-party tools have made in the TOOLOPTIONS section of the .TXD file for the dictionary.
%DictionaryDescription	A string containing a short description of the dictionary.
%DictionaryVersion	A string containing the current dictionary version number.
%DictionaryTimeCreated	The dictionary creation time (Clarion standard time).
%DictionaryDateCreated	The dictionary creation date (Clarion standard date).
%DictionaryTimeChanged	The time dictionary was last changed (Clarion standard time).
%DictionaryDateChanged	The date dictionary was last changed (Clarion standard date).
%File	Contains all file declarations in the .DCT file. Multi-valued. Dependent on %DictionaryFile.
%Program	The name of the PROGRAM module.
%GlobalData	The labels of all global variable declarations made through the Global Data button on the Global Settings window. Multi-valued.

%GlobalDataInDictionary	Contains 1 if the global variable is defined within the dictionary. Dependent on %GlobalData.
%GlobalDataStatement	The variable's declaration statement (data type and all attributes). Dependent on %GlobalData.
%GlobalDataLast	Contains 1 if the "Generate Last" option is set for %FileUsage = GLOBAL. Dependent on %GlobalData.
%GlobalDataLevel	The indent level for formatting complex data structures. Dependent on %GlobalData.
%GlobalDataUserOptions	A string containing the entries the user made in the User Options text box on the Options tab for the global data group defined in the dictionary.
%Module	The names of all source code modules other than the PROGRAM module. Multi-valued.
%CreateLocalMap	Contains 1 if the application should create local MAP structures instead of a single global MAP.
%QuickProcedure	The name of the procedure type a #UTILITY with the WIZARD attribute is creating.
%EditProcedure	The name of the procedure being edited in context.
%EditFilename	The name of the temporary file that the procedure being edited in context is generated into.
%Procedure	The names of all procedures in the application. Multi-valued.

Symbols Dependent on %File

%File	Contains all file declarations in the .DCT file. Multi-valued. Dependent on %DictionaryFile.
%FileIdent	The internal file number assigned by the Dictionary Editor and displayed in the IDENT in a .TXD file.
%FilePrefix	Contents of the PRE attribute (the file prefix).
%FileDescription	A short description of the file.
%FileType	Contains FILE, VIEW, or ALIAS.
%FileDriver	Contents of the DRIVER attribute first parameter.
%FileDriverParameter	Contents of the DRIVER attribute second parameter.
%FileName	Contents of the FILE statement's NAME attribute.
%FileOwner	Contents of the OWNER attribute.
%FileCreate	Contains 1 if the file has the CREATE attribute.
%FileReclaim	Contains 1 if the file has the RECLAIM attribute.
%FileOEM	Contains a 1 if the file has the OEM attribute.
%FileEncrypt	Contains 1 if the file has the ENCRYPT attribute.
%FileBindable	Contains 1 if the file has the BINDABLE attribute.
%FileLongDesc	A long description of the file.
%File32BitOnly	Contains 1 if the file can only be used in 32-bit applications.
%FileStruct	The FILE statement (the label and all attributes).
%FileStructEnd	The keyword END.
%FileStructRec	The RECORD statement (including label and any attributes).
%FileStructRecEnd	The keyword END.
%FileStatement	Contains the FILE statement's attributes (only).
%FileThreaded	Contains 1 if the file has the THREAD attribute.

%FileExternal	Contains 1 if the file has the EXTERNAL attribute.
%FileExternalModule	Contents of the file's EXTERNAL attribute parameter.
%FilePrimaryKey	The label of the file's primary key.
%FileQuickOptions	A comma-delimited string containing the choices the user made on the Options tab for the file.
%FileTriggerBeforeInsert	Contains the contents of the trigger condition.
%FileTriggerAfterInsert	Contains the contents of the trigger condition.
%FileTriggerBeforeUpdate	Contains the contents of the trigger condition.
%FileTriggerAfterUpdate	Contains the contents of the trigger condition.
%FileTriggerBeforeDelete	Contains the contents of the trigger condition.
%FileTriggerAfterDelete	Contains the contents of the trigger condition.
%FileTriggerDataInitialValue	Contains the contents of the Initial Value of a Trigger Data Element.
%FileUserOptions	A string containing the entries the user made in the User Options text box on the Options tab for the file.
%ViewFilter	Contents of the FILTER attribute.
%ViewStruct	The VIEW statement (including the label and all attributes).
%ViewStructEnd	The keyword END.
%ViewStatement	The VIEW statement's attributes (only).
%ViewPrimary	The label of the VIEW's primary file.
%ViewPrimaryFields	The labels of all fields in the VIEW from the primary file. Multi-valued.
%ViewPrimaryField	Dependent on %ViewPrimaryFields. Contains the label of a field in the VIEW from the primary file.

%ViewFiles	The labels of all files in the VIEW. Multi-valued.
%AliasFile	The label of the ALIASed file.
%Field	The labels of all fields in the file (including MEMO fields). Multi-valued.
%Key	The labels of all keys and indexes for the file. Multi-valued.
%Relation	The labels of all files that are related to the file. Multi-Valued.

Symbols Dependent on %ViewFiles

%ViewFiles	The labels of all files in the VIEW. Multi-valued. Dependent on %File.
%ViewFileStruct	The JOIN statement for a secondary file in the VIEW.
%ViewFileStructEnd	The keyword END.
%ViewFile	Contains the label of the file.
%ViewJoinedTo	The label of the file to which the file is JOINed.
%ViewFileFields	The labels of all fields in the file used in the VIEW. Multi-valued.
%ViewFileField	Contains the label of the field in the file used in the VIEW. Dependent on %ViewFileFields.

Symbols Dependent on %Field

%Field	The labels of all fields in the file (including MEMO fields). Multi-valued.Dependent on %File.
%FieldIdent	The internal field number assigned by the Dictionary Editor and displayed in the IDENT in a .TXD file.
%FieldDescription	A short description of the field.
%FieldLongDesc	A long description of the field.
%FieldFile	The label of the file containing the field.
%FieldID	Label of the field without prefix.
%FieldDisplayPicture	Default display picture.
%FieldRecordPicture	STRING field storage definition picture.
%FieldDimension1	Maximum value of first array dimension.
%FieldDimension2	Maximum value of second array dimension.
%FieldDimension3	Maximum value of third array dimension.
%FieldDimension4	Maximum value of fourth array dimension.
%FieldHelpID	Contents of the HLP attribute.
%FieldName	Contents of the field's NAME attribute.
%FieldRangeLow	The lower range of valid values for the field.
%FieldRangeHigh	The upper range of valid values for the field.
%FieldType	Data type of the field.
%FieldPlaces	Number of decimal places for the field.
%FieldMemoSize	Maximum size of the MEMO.
%FieldMemoImage	Contains 1 if the MEMO has a BINARY attribute.

%FieldInitial	Initial value for the field.
%FieldLookup	File to access to validate this field's value.
%FieldStruct	The field's declaration statement (label , data type, and all attributes).
%FieldStatement	The field's declaration statement (data type and all attributes).
%FieldHeader	The field's default report column header.
%FieldPicture	Default display picture.
%FieldJustType	Contains L, R, C, or D for the field's justification.
%FieldJustIndent	The justification offset amount.
%FieldFormatWidth	The default width for the field's ENTRY control.
%FieldScreenControl	The field's screen control as defined in the data dictionary. Multi-valued.
%FieldScreenControlWidth	Either the default from the runtime library, or the explicitly set width in the data dictionary for the field's screen control. Dependent on %FieldScreenControl.
%FieldScreenControlHeight	Either the default from the runtime library, or the explicitly set height in the data dictionary for the field's screen control. Dependent on %FieldScreenControl.
%FieldReportControl	The field's report control as defined in the data dictionary. Multi-valued.
%FieldReportControlWidth	Either the default from the runtime library, or the explicitly set width in the data dictionary for the field's report control. Dependent on %FieldReportControl.
%FieldReportControlHeight	Either the default from the runtime library, or the explicitly set height in the data dictionary for the field's report control. Dependent on %FieldReportControl.
%FieldValidation	The choice the user made on the Validity Checks tab for the field. Can contain either NONZERO, INRANGE, BOOLEAN, INFILE, or INLIST.
%FieldChoices	The choices the user entered for a Must Be In List field (%FieldValidation contains INLIST). Multi-valued.
%FieldValues	The choices the user entered to override the VALUE attribute for the %FieldChoices symbol. Multi-valued.

%FieldTrueValue	The true choice the user entered to override the VALUE attribute (%FieldValidation contains BOOLEAN).
%FieldFalseValue	The false choice the user entered to override the VALUE attribute (%FieldValidation contains BOOLEAN).
%FieldDisplayChoices	The choices the user entered in the display override field for a Must Be In List field (%FieldValidation contains INLIST). Multi-valued.
%FieldQuickOptions	A comma-delimited string containing the choices the user made on the Options tab for the field.
%FieldUserOptions	A string containing the entries the user made in the User Options text box of the Options tab for the field.
%FieldToolOptions	A string containing the entries third-party tools have made in the TOOLOPTIONS section of the .TXD file for the field.

Symbols Dependent on %Key and %KeyOrder

%Key	The labels of all keys and indexes for the file. Multi-valued.
%KeyIdent	The internal key number assigned by the Dictionary Editor and displayed in the IDENT in a .TXD file.
%KeyDescription	A short description of the key.
%KeyLongDesc	A long description of the key.
%KeyFile	The label of the file to which the key belongs.
%KeyID	The label of the key (without prefix).
%KeyIndex	Contains KEY, INDEX, or DYNAMIC.
%KeyName	Contents of the key's NAME attribute.
%KeyAuto	Contains the label of the auto-incrementing field.
%KeyDuplicate	Contains 1 if the key has the DUP attribute.
%KeyExcludeNulls	Contains 1 if the key has the OPT attribute.
%KeyNoCase	Contains 1 if the key has the NOCASE attribute.
%KeyPrimary	Contains 1 if the key is the file's primary key.
%KeyStruct	The key's declaration statement (label and all attributes).
%KeyStatement	The key's attributes (only).
%KeyField	The labels of all component fields of the key. Multi-valued.
%KeyFieldSequence	Contains ASCENDING or DESCENDING. Dependent on %Keyfield.
%KeyQuickOptions	A comma-delimited string containing the choices the user made on the Options tab for the key.
%KeyUserOptions	A string containing the entries the user made in the User Options text box on the Options tab for the key.
%KeyToolOptions	A string containing the entries third-party tools have made in the TOOLOPTIONS section of the .TXD file for the key.
%KeyOrder	The labels of all keys, indexes, and orders for the file. Multi-valued.

%KeyOrderIdent	The internal key number assigned by the Dictionary Editor and displayed in the IDENT in a .TXD file.
%KeyOrderDescription	A short description of the key.
%KeyOrderLongDesc	A long description of the key.
%KeyOrderFile	The label of the file to which the key belongs.
%KeyOrderID	The label of the key (without prefix).
%KeyOrderIndex	Contains KEY, INDEX, or DYNAMIC.
%KeyOrderName	Contents of the key's NAME attribute.
%KeyOrderAuto	Contains the label of the auto-incrementing field.
%KeyOrderDuplicate	Contains 1 if the key has the DUP attribute.
%KeyOrderExcludeNulls	Contains 1 if the key has the OPT attribute.
%KeyOrderNoCase	Contains 1 if the key has the NOCASE attribute.
%KeyOrderPrimary	Contains 1 if the key is the file's primary key.
%KeyOrderStruct	The key's declaration statement (label and all attributes).
%KeyOrderStatement	The key's attributes (only).
%KeyOrderQuickOptions	A comma-delimited string containing the choices the user made on the Options tab for the key.
%KeyOrderUserOptions	A string containing the entries the user made in the User Options text box on the Options tab for the key.
%KeyOrderToolOptions	A string containing the entries third-party tools have made in the TOOLOPTIONS section of the .TXD file for the key.

Symbols Dependent on %Relation

%Relation	The labels of all files that are related to the file. Multi-Valued.
%RelationPrefix	The prefix of the related file.
%RelationAlias	Contains the name of the file being aliased if the current file named in %Relation is an alias.
%FileRelationType	Contains 1:MANY or MANY:1.
%RelationKey	The label of the related file's linking key.
%RelationPrimaryKey	The label of the related file's primary key.
%FileKey	The label of the file's linking key.
%RelationDriver	The label of the related file's driver.
%RelationConstraintDelete	May contain: RESTRICT, CASCADE, or CLEAR.
%RelationConstraintUpdate	May contain: RESTRICT, CASCADE, or CLEAR.
%RelationKeyField	The labels of all linking fields in the related file's key. Multi-valued.
%RelationKeyFieldLink	The label of the linking field in the file's key. Dependent on %RelationKeyField.
%FileKeyField	The labels of all linking fields in the file's key. Multi-valued.
%FileKeyFieldLink	The label of the linking field in the related file's key. Dependent on %FileKeyField.
%RelationQuickOptions	A comma-delimited string containing the choices the user made on the Options tab for the relation.
%RelationUserOptions	A string containing the entries the user made in the User Options text box on the Options tab for the relation.
%RelationToolOptions	A string containing the entries third-party tools have made in the TOOLOPTIONS section of the .TXD file for the relation.

Symbols Dependent on %Module

%Module	Thes names of all source code modules other than the PROGRAM module. Multi-valued
%ModuleDescription	Contains the description of the module.
%ModuleLanguage	Contains the module target language.
%ModuleTemplate	The name of the Module Template used to generate the module.
%ModuleChanged	Contains 1 if anything in the module has changed since the last source generation.
%ModuleExternal	Contains 1 if the module is external. (not generated by Clarion for Windows).
%ModuleReadOnly	Contains 1 if the module is read only.
%ModuleExtension	The file extension for the module.
%ModuleBase	The name of the module (without extension).
%ModuleInclude	The file to INCLUDE in the program MAP containing the module's prototypes.
%ModuleProcedure	The names of all procedures in the module. Multi-valued.
%ModuleData	The labels of all module variable declarations made through the Data button on the Module Properties window. Multi-valued.
%ModuleDataStatement	The variable's declaration statement (data type and all attributes). Dependent on %ModuleData.
%ModuleDataUserOptions	A string containing the entries the user made in the User Options text box on the Options tab for the variable. Dependent on %ModuleData.

Symbols Dependent on %Procedure

%Procedure	The names of all procedures in the application. Multi-valued.
%ProcedureToDo	Contains 1 if the procedure is "ToDo." Dependent on %Procedure.
%ProcedureType	Contains PROCEDURE or FUNCTION. Dependent on %Procedure.
%ProcedureReturnType	The data type returned, if the procedure is a FUNCTION. Dependent on %Procedure.
%ProcedureDateCreated	The procedure creation date (a Clarion standard date). Dependent on %Procedure.
%ProcedureDateChanged	The date the procedure was last changed (a Clarion standard date). Dependent on %Procedure.
%ProcedureTimeCreated	The time the procedure was created (a Clarion standard time). Dependent on %Procedure.
%ProcedureTimeChanged	The time the procedure was last changed (a Clarion standard time). Dependent on %Procedure.
%ProcedureReadOnly	Contains 1 if the procedure is read only. Dependent on %Procedure.
%ProcedureIsGlobal	Contains 1 if the procedure is to be declared in the global MAP. Dependent on %Procedure.
%Prototype	The procedure's prototype for the MAP structure. Dependent on %Procedure.
%ProcedureTemplate	The name of the Procedure Template used to generate the procedure. Dependent on %Procedure.
%ProcedureDescription	A short description of the procedure. Dependent on %Procedure.
%ProcedureCategory	The procedure's category. Dependent on %Procedure.
%ProcedureExported	Contains 1 if the procedure is in a DLL and is callable from outside the DLL. Dependent on %Procedure.

%ProcedureLongDescription	A long description of the procedure. Dependent on %Procedure.
%ProcedureLanguage	The target language the procedure template generates. Dependent on %Procedure.
%ProcedureCalled	The names of all procedures listed by the Procedures button on the Procedure Properties window. Multi-valued. Dependent on %Procedure.
%LocalData	The labels of all local variable declarations made through the Data button on the Procedure Properties window. Multi-valued. Dependent on %Procedure.
%LocalDataStatement	The variable's declaration statement (data type and all attributes). Dependent on %LocalData.
%LocalDataDescription	A short description of the field. Dependent on %LocalData.
%LocalDataHeader	The field's default report column header. Dependent on %LocalData.
%LocalDataPicture	Default display picture. Dependent on %LocalData.
%LocalDataJustType	Contains L, R, C, or D for the field's justification. Dependent on %LocalData.
%LocalDataJustIndent	The justification offset amount. Dependent on %LocalData.
%LocalDataFormatWidth	The default width for the field's ENTRY control. Dependent on %LocalData.
%LocalDataUserOptions	A string containing the entries the user made in the User Options text box on the Options tab for the local variable. Dependent on %LocalData.
%ActiveTemplate	The names of all control templates used in the procedure. Multi-valued. Dependent on %Procedure.
%ActiveTemplateInstance	The instance numbers of all control templates used in the procedure. Multi-valued. Dependent on %ActiveTemplate.
%ActiveTemplateOwnerInstance	The instance number of the extension template which auto-populated the current extension template into a procedure through use of the parameter to the APPLICATION attribute on a #EXTENSION statement. Dependent on %ActiveTemplateInstance.

%ActiveTemplateParentInstance	The instance number of the control template's parent control template. This is the control template that it is "attached" to. Dependent on %ActiveTemplateInstance.
%ActiveTemplatePrimaryInstance	The instance number of the control template's primary control template. This is the first control template in a succession of multiple related control templates. Dependent on %ActiveTemplateInstance.
%ActiveTemplateInstanceDescription	The description of the control template. Dependent on %ActiveTemplateInstance.
%ActiveTemplateType	The type of all control templates used in the procedure. Multi-valued. Dependent on %ActiveTemplate. Returns CODE, EXTENSION or CONTROL.

Window Control Symbols

%Window	The label of the procedure's window. Dependent on %Procedure.
%WindowStatement	The WINDOW or APPLICATION declaration statement (and all attributes). Dependent on %Window.
%MenuBarStatement	The MENUBAR declaration statement (and all attributes). Dependent on %Window.
%ToolbarStatement	The TOOLBAR declaration statement (and all attributes). Dependent on %Window.
%WindowEvent	All field-independent events, as listed in the EQUATES.CLW file (without EVENT: prepended). Multi-valued. Dependent on %Window.
%Control	The field equate labels of all controls in the window. Multi-valued. Dependent on %Window.
%ControlUse	The control's USE variable (not field equate). Dependent on %Control.
%ControlStatement	The control's declaration statement (and all attributes). This may contain multiple lines of code if the declaration is too long to fit on a single line. Dependent on %Control.
%ControlUnsplitStatement	The control's declaration statement (and all attributes) without line spliting. Dependent on %Control.
%ControlType	The type of control (MENU, ITEM, ENTRY, BUTTON, etc.). Dependent on %Control.
%ControlTemplate	The name of the control template which populated the control onto the window. Dependent on %Control.
%ControlTool	Contains 1 if the control is in a TOOLBAR. Dependent on %Control.
%ControlParent	Contains the field equate label of the control's parent, if it is in a control structure (OPTION, GROUP, etc.). Dependent on %Control.
%ControlParentType	Contains the type of control of the control's parent, if it is in a control structure (OPTION, GROUP, etc.). Dependent on %ControlParent.

%ControlParentTab	Contains the field equate label of the TAB on which the control has been placed. Dependent on %Control.
%ControlParameter	The control's parameter (the value ion the parentheses). Dependent on %Control.
%ControlHasIcon	Contains 1 if the LIST or COMBO control contains any icon attributes. Dependent on %Control.
%ControlHasColor	Contains 1 if the LIST or COMBO control contins any color attributes. Dependent on %Control.
%ControlHasStyle	Contains 1 if the LIST or COMBO control contains any style attributes. Dependent on %Control.
%ControlHasTip	Contains 1 if the LIST or COMBO control contains any tool tip attributes. Dependent on %Control.
%ControlHasTree	Contains 1 if the LIST or COMBO control contains any tree attributes. Dependent on %Control.
%ControlDefaultWidth	The control's estimated default width. Dependent on %Control.
%ControlDefaultHeight	The control's estimated default height. Dependent on %Control.
%ControlMenu	Contains 1 if the control is in a MENUBAR. Dependent on %Control.
%ControlToolBar	Contains the TOOLBAR declaration statement (and all attributes) if the control is the first control in a toolbar. Dependent on %Control.
%ControlMenuBar	Contains the MENUBAR declaration statement (and all attributes) if the control is the first control in a menu. Dependent on %Control.
%ControlIndent	The control declaration's indentation level in the generated data structure. Dependent on %Control.
%ControlInstance	The instance number of the control template which populated the control onto the window. Dependent on %Control.

%ControlOriginal	The original field equate label of the control as listed in the control template from which it came. Dependent on %Control.
%ControlFrom	The FROM attribute of a LIST or COMBO control. Dependent on %Control.
%ControlAlert	All ALRT attributes for the control. Multi-valued. Dependent on %Control.
%ControlEvent	All field-specific events appropriate for the control, as listed in the EQUATES.CLW file (without the EVENT: prepended). Multi-valued. Dependent on %Control.
%ControlField	All fields populated into the LIST, COMBO, or SPIN control. Multi-valued. Dependent on %Control.
%ControlFieldHasIcon	Contains 1 if the field in the LIST or COMBO control is formatted to have an icon. Dependent on %ControlField.
%ControlFieldHasColor	Contains 1 if the field in the LIST or COMBO control is formatted to have colors. Dependent on %ControlField.
%ControlFieldHasTree	Contains 1 if the field in the LIST or COMBO control is formatted to be a tree. Dependent on %ControlField.
%ControlFieldHasLocator	Contains 1 if the field in the LIST or COMBO control is formatted to be a locator. Dependent on %ControlField.
%ControlFieldHasTip	Contains 1 if the field in the LIST or COMBO control is formatted to have a tool tip. Dependent on %ControlField.
%ControlFieldDefaultTip	A string value that contains value of the column default tip if it is given, or an empty string otherwise.
%ControlFieldPicture	Contains the picture token of the field in the LIST or COMBO control. Dependent on %ControlField.
%ControlFieldHeader	Contains the heading text of the field in the LIST or COMBO control. Dependent on %ControlField.
%ControlFieldFormat	Contains the portion of the FORMAT attribute string that applies to the field in the LIST or COMBO control. Dependent on %ControlField.

Report Control Symbols

%Report	The label of the procedure's report. Dependent on %Procedure.
%ReportStatement	The REPORT declaration statement (and all attributes). Dependent on %Report.
%ReportControl	The field equate labels of all controls in the report. Multi-valued. Dependent on %Report.
%ReportControlUse	The control's USE variable (not field equate). Dependent on %ReportControl.
%ReportControlStatement	The control's declaration statement (and all attributes). Dependent on %ReportControl.
%ReportControlType	The type of control (MENU, ITEM, ENTRY, BUTTON, etc.). Dependent on %ReportControl.
%ReportControlTemplate	The name of the control template which populated the control onto the report. Dependent on %ReportControl.
%ReportControlIndent	The control declaration's indentation level in the generated data structure. Dependent on %ReportControl.
%ReportControlInstance	The instance number of the control template which populated the control onto the report. Dependent on %ReportControl.
%ReportControlOriginal	The original field equate label of the control as listed in the control template from which it came. Dependent on %ReportControl.
%ReportControlLabel	The label of the report STRING control. Dependent on %ReportControl.
%ReportControlField	All fields populateed into the LIST, COMBO, or SPIN control. Multi-valued. Dependent on %ReportControl.

Formula Symbols

%Formula	The label of the result field for each formula. Multi-valued. Dependent on %Procedure.
%FormulaDescription	A description of the formula.
%FormulaClass	An identifier for the position in generated source to place the formula.
%FormulaInstance	The control template instance number for a formula whose class has been declared in a control template.
%FormulaExpression	The expression to conditionally evaluate or assign to the result field for each formula. Multi-valued. Dependent on %Formula.
%FormulaExpressionType	Contains =, IF, ELSE, CASE, or OF. Dependent on %FormulaExpression.
%FormulaExpressionTrue	Contains the line number of the true expression in the generated formula. Dependent on %FormulaExpression.
%FormulaExpressionFalse	Contains the line number of the false expression in the generated formula. Dependent on %FormulaExpression.
%FormulaExpressionOf	Contains the line number of the OF expression in the generated formula. Dependent on %FormulaExpression.
%FormulaExpressionCase	Contains the line number of the assignment in the generated formula. Dependent on %FormulaExpression.

File Schematic Symbols

%Primary	The label of a Primary file listed in the procedure's File Schematic for the procedure or a control template used in the procedure.
%PrimaryKey	The label of the access key for the primary file. Dependent on %Primary.
%PrimaryInstance	The control template instance number for which the file is primary. Dependent on %Primary.
%Secondary	The labels of all Secondary files listed in the File Schematic for the procedure or a control template used in the procedure. Multi-valued. Dependent on %Primary.
%SecondaryTo	The label of the Secondary or Primary file to which the Secondary file is related (thefile"above" it as listed in the procedure's File Schematic). Dependent on %Secondary.
%SecondaryType	Contains 1:MANY or MANY:1. Dependent on %Secondary.
%SecondaryCustomJoin	Contains 1 if the JOIN is a custom join. Dependent on %Secondary.
%SecondaryCustomText	Contains the text defining the custom join. Dependent on %Secondary.
%SecondaryInner	Contains a 1 if the custom relationship uses an INNER join.
%OtherFiles	The labels of all Other Data files listed for the procedure. Multi-valued.

File Driver Symbols

%Driver	The names of all registered file drivers.
%DriverDLL	The name of the driver's .DLL file. Dependent on %Driver.
%DriverLIB	The name of the driver's .LIB file. Dependent on %Driver.
%DriverDescription	A description of the file driver. Dependent on %Driver.
%DriverCreate	Contains 1 if the driver supports the CREATE attribute. Dependent on %Driver.
%DriverOwner	Contains 1 if the driver supports the OWNER attribute. Dependent on %Driver.
%DriverEncrypt	Contains 1 if the driver supports the ENCRYPT attribute. Dependent on %Driver.
%DriverReclaim	Contains 1 if the driver supports the RECLAIM attribute. Dependent on %Driver.
%DriverMaxKeys	The maximum number of keys the driver supports for each data file. Dependent on %Driver.
%DriverUniqueKey	Contains 1 if the driver supports unique (no DUP attribute) keys. Dependent on %Driver.
%DriverRequired	Contains 1 if the driver supports the OPT attribute. Dependent on %Driver.
%DriverMemo	Contains 1 if the driver supports MEMO fields. Dependent on %Driver.
%DriverBinMemo	Contains 1 if the driver supports the BINARY attribute on MEMO fields. Dependent on %Driver.
%DriverSQL	Contains 1 if the driver is an SQL driver. Dependent on %Driver.
%DriverOEM	Contains 1 if the driver supports the OEM attribute. Depenent on %Driver.
%DriverType	All data types supported bythe driver. Multi-valued. Dependent on %Driver.
%DriverOpcode	All operations supported by the driver. Multi-valued. Dependent on %Driver.

Miscellaneous Symbols

%ConditionalGenerate	Contains 1 if the Conditional Generation box is checked on the Application Options window.
%CWRoot	Contains the value of the "Root" key found in the CW section of the WIN.INI file.
%CWVersion	Contains the current version release number of Clarion for Windows. For example, for the first release of version 2.0, this contains 2000.
%Null	Contains nothing. This is used for comparison to detect empty symbols.
%True	Contains 1.
%False	Contains an empty string ('').
%EOF	Contains the value that flags the end of file when reading a file with #READ.
%BytesOutput	Contains the number of bytes written to the current output file. This can be used to detect empty embed points (if no bytes were written, it contained nothing).
%OutputOffset	Same as %BytesOutput.
%OutputIndent	Contains the implicit indentation level.
%OutputColumn	Current column in the output file. Could be used for aligning to different columns.
%CommentColumn	Contains the column where comments (#<) are generated.
%EmbedID	Contains the current embed point's identifying symbol.
%EmbedDescription	The current embed point's description.
%EmbedParameters	The current embed point's current instance, as a comma-delimited list.
%UtilityArguments	Set to the parameter when executing a Utility template from a DDE call.
%Win32	Contains 1 if #RUNDLL with the WIN32 attribute can be executed.

5 - Annotated Examples

Procedure Template: Window

The *Window* Procedure template is the generic template that creates any window handling procedure. Since most (if not all) procedures in a Windows application have a window, the type of code this template generates forms the basis of the generated source code for most procedures.

The Window template is also the fundamental template upon which all the other Procedure templates are built. For example, the Browse template is actually a Window template with BrowseBox and BrowseUpdateButtons Control templates pre-defined for the procedure.

The following template language code is all the code for the Window Template:

NOTE:
For this and all other code examples in this book, the template line continuation character (%|) is used to split code lines that are too long to fit on the page. In the template files on disk these characters are not (and should not be) used to continue a template code line; they are used here only for readability. These templates are from the Clarion template chain, not ABC.

```
#PROCEDURE(Window,'Generic Window Handler'),WINDOW,HLP('~TPLProcWindow')
#LOCALDATA
LocalRequest         LONG,AUTO
OriginalRequest      LONG,AUTO
LocalResponse        LONG,AUTO
WindowOpened         LONG
WindowInitialized    LONG
ForceRefresh         LONG,AUTO
#ENDLOCALDATA
#CLASS('Procedure Setup','Upon Entry into the Procedure')
#CLASS('Before Lookups','Refresh Window ROUTINE, before lookups')
#CLASS('After Lookups','Refresh Window ROUTINE, after lookups')
#CLASS('Procedure Exit','Before Leaving the Procedure')
#PROMPT('&Parameters:', @s255), %Parameters
#ENABLE(%ProcedureType='FUNCTION')
  #PROMPT('Return Value:',FIELD),%ReturnValue
#ENDENABLE
#PROMPT('Window Operation Mode:',DROP('Use WINDOW setting|Normal|MDI|Modal')) %|
         ,%WindowOperationMode
#ENABLE(%INIActive)
  #BOXED('INI File Settings')
    #PROMPT('Save and Restore Window Location',CHECK) %|
             ,%INISaveWindow,DEFAULT(1),AT(10,,150)
  #ENDBOXED
#ENDENABLE
```

```
#AT(%CustomGlobalDeclarations)
  #INSERT(%StandardGlobalSetup)
#ENDAT
#INSERT(%StandardWindowCode)
```

This code starts with the #PROCEDURE statement, which names the Procedure template and indicates that it will have a WINDOW (or APPLICATION) structure, but no REPORT. The #LOCALDATA section defines six local variables that generate automatically as part of the procedure. These are common local variables for most generated procedures.

The #CLASS statements define the formula classes for the Formula Editor. These identify the source code positions at which formulas generate.

The #PROMPT statements create the prompts on the **Procedure Properties** window. The first allows the programmer to name the parameters passed into the procedure. The #ENABLE structure enables its #PROMPT only when the %ProcedureType symbol contains "FUNCTION." This occurs only when the **Prototype** prompt (standard on all procedures) contains a procedure prototype with a return data type.

The next #PROMPT allows the programmer to override the window's operation mode as specified on the WINDOW structure. The next #ENABLE structure enables its #BOXED #PROMPT only when the %INIActive symbol contains a value. This symbol comes from a check box on the Global Settings window.

The #AT structure calls the %StandardGlobalSetup #GROUP. This contains code to determine if the procedure is using any .VBX controls. It so, they are added to the list of files to ship with the application that generates into the *ProgramName*.SHP file.

You will note that none of these statements generates any target language (Clarion) source code other than the six variable declarations. The last #INSERT statement places all the code the %StandardWindowCode #GROUP generates at the end of these statements. This is the #GROUP that handles all the source generation for the template.

See Also:

%StandardWindowCode #GROUP

%StandardWindowHandling #GROUP

%StandardAcceptedHandling #GROUP

%StandardControlHandling #GROUP

%StandardWindowCode #GROUP

This #GROUP actually generates all the source code for the Window template. This includes all the local data declarations, standard window handling code, and provides all the "hooks" for all the control and extension templates to attach into the generated procedure.

```
#GROUP(%StandardWindowCode)
#IF(NOT %Window)
  #ERROR(%Procedure & ' Error: No Window Defined!')
  #RETURN
#ENDIF
#DECLARE(%FirstField)
#DECLARE(%LastField)
#DECLARE(%ProgressWindowRequired)
#INSERT(%FieldTemplateStandardButtonMenuPrompt)
#INSERT(%FieldTemplateStandardEntryPrompt)
#INSERT(%FieldTemplateStandardCheckBoxPrompt)
#EMBED(%GatherSymbols,'Gather Template Symbols'),HIDE
#INSERT(%FileControlInitialize)
%Procedure %ProcedureType%Parameters

#FOR(%LocalData)
%[20]LocalData %LocalDataStatement
#ENDFOR
#INSERT(%StandardWindowGeneration)
#IF(%ProgressWindowRequired)
#INSERT(%StandardProgressWindow)
#ENDIF
  CODE
  PUSHBIND
  #EMBED(%ProcedureInitialize,'Initialize the Procedure')
  LocalRequest = GlobalRequest
  OriginalRequest = GlobalRequest
  LocalResponse = RequestCancelled
  ForceRefresh = False
  CLEAR(GlobalRequest)
  CLEAR(GlobalResponse)
  #EMBED(%ProcedureSetup,'Procedure Setup')
  IF KEYCODE() = MouseRight
    SETKEYCODE(0)
  END
  #INSERT(%StandardFormula,'Procedure Setup')
  #INSERT(%FileControlOpen)
  #INSERT(%StandardWindowOpening)
  #EMBED(%PrepareAlerts,'Preparing Window Alerts')
  #EMBED(%BeforeAccept,'Preparing to Process the Window')
  #MESSAGE('Accept Handling',3)
  ACCEPT
```

```
    #EMBED(%AcceptLoopBeforeEventHandling,'Accept Loop, Before CASE EVENT() handling')
    CASE EVENT()
    #EMBED(%EventCaseBeforeGenerated,'CASE EVENT() structure, before generated code')
    #INSERT(%StandardWindowHandling)
    #EMBED(%EventCaseAfterGenerated,'CASE EVENT() structure, after generated code')
    END
    #EMBED(%AcceptLoopAfterEventHandling,'Accept Loop, After CASE EVENT() handling')
    #SUSPEND
    #?CASE ACCEPTED()
    #INSERT(%StandardAcceptedHandling)
    #?END
    #RESUME
    #EMBED(%AcceptLoopBeforeFieldHandling,'Accept Loop, Before CASE FIELD() handling')
    #SUSPEND
    #?CASE FIELD()
    #EMBED(%FieldCaseBeforeGenerated,'CASE FIELD() structure, before generated code')
    #INSERT(%StandardControlHandling)
    #EMBED(%FieldCaseAfterGenerated,'CASE FIELD() structure, after generated code')
    #?END
    #RESUME
    #EMBED(%AcceptLoopAfterFieldHandling,'Accept Loop, After CASE FIELD() handling')
  END
  DO ProcedureReturn
!----------------------------------------------------------------------------
ProcedureReturn ROUTINE
!|
!| This routine provides a common procedure exit point for all template
!| generated procedures.
!|
!| First, all of the files opened by this procedure are closed.
!|
!| Next, if it was opened by this procedure, the window is closed.
!|
!| Next, GlobalResponse is assigned a value to signal the calling procedure
!| what happened in this procedure.
!|
!| Next, we replace the BINDings that were in place when the procedure
initialized
!| (and saved with PUSHBIND) using POPBIND.
!|
#IF(%ReturnValue)
!| Finally, we return to the calling procedure, passing %ReturnValue back.
#ELSE
!| Finally, we return to the calling procedure.
#ENDIF
!|
  #INSERT(%FileControlClose)
  #INSERT(%StandardWindowClosing)
  #EMBED(%EndOfProcedure,'End of Procedure')
  #INSERT(%StandardFormula,'Procedure Exit')
  IF LocalResponse
    GlobalResponse = LocalResponse
  ELSE
    GlobalResponse = RequestCancelled
  END
  POPBIND
  #IF(%ProcedureType='FUNCTION')
  RETURN(%ReturnValue)
  #ELSE
  RETURN
  #ENDIF
!----------------------------------------------------------------------------
InitializeWindow ROUTINE
!|
!| This routine is used to prepare any control templates for use. It should be called
!| once per procedure.
!|
  #EMBED(%WindowInitializationCode,'Window Initialization Code')
  DO RefreshWindow
!----------------------------------------------------------------------------
RefreshWindow ROUTINE
!|
!| This routine is used to keep all displays and control templates current.
!|
  IF %Window{Prop:AcceptAll} THEN EXIT.
  #EMBED(%RefreshWindowBeforeLookup,'Refresh Window routine, before lookups')
  #INSERT(%StandardFormula,'Before Lookups')
  #INSERT(%StandardSecondaryLookups)
  #INSERT(%StandardFormula,'After Lookups')
  #EMBED(%RefreshWindowAfterLookup,'Refresh Window routine, after lookups')
  #EMBED(%RefreshWindowBeforeDisplay,'Refresh Window routine, before DISPLAY()')
  DISPLAY()
  ForceRefresh = False
!----------------------------------------------------------------------------
SyncWindow ROUTINE
  #EMBED(%SyncWindowBeforeLookup,'Sync Record routine, before lookups')
  #INSERT(%StandardFormula,'Before Lookups')
  #INSERT(%StandardSecondaryLookups)
  #INSERT(%StandardFormula,'After Lookups')
  #EMBED(%SyncWindowAfterLookup,'Sync Record routine, after lookups')
!----------------------------------------------------------------------------
#EMBED(%ProcedureRoutines,'Procedure Routines')
#EMBED(%LocalProcedures,'Local Procedures'),HIDE
```

This starts with the required #GROUP statement which identifies the group for use in #INSERT statements.

The #IF(NOT %Window) error check detects whether the programmer has forgotten to create a window for the procedure. The #ERROR statement alerts the programmer to the mistake and #RETURN immediately aborts any further source generation for the procedure. The #DECLARE statements declare two template symbols for internal use by other #GROUPs that are called to generate source for the procedure, and a "flag" that determines whether a "progress" window is required by the procedure.

The next three #INSERT statements insert #GROUPs that contain #FIELD structures to define the standard prompts that appear on the **Actions** tab for BUTTON, ENTRY and CHECK controls placed on the procedure's window. These prompts allow the programmer to specify the standard actions these controls can take from this procedure.

The %GatherSymbols #EMBED statement has the HIDE attribute. This means that it will not appear in the list of available embed points for the programmer to insert code, making the embed point only available for internal use (for Code, Control, or Extension templates to generate code into).

The #INSERT(%FileControlInitialize) statement inserts a #GROUP that updates the symbols that keep track of the files used in the application with the files used by this procedure.

The %Procedure %ProcedureType%Parameters statement generates the first Clarion language source code for the procedure. It generates the procedure's PROCEDURE statement, with or without a parameter list, as appropriate.

The #FOR(%LocalData) loop generates all the local variable declarations for the procedure. The %[20]LocalData syntax means that the %LocalData symbol expands to fill at least 20 spaces before the %LocalDataStatement symbol expands. This aligns the data types for each variable declaration starting in column 22.

The #INSERT(%StandardWindowGeneration) statement generates the procedure's WINDOW or APPLICATION data structure. This #GROUP also contains two #EMBED statements that allow the programmer to embed code either before or after the window structure.

The #IF(%ProgressWindowRequired) statement conditionally #INSERTs the %StandardProgressWindow group, which generates the ProgressWindow WINDOW structure for the procedure.

Next, the CODE statement generates, to begin the procedure's executable code section, which starts with a PUSHBIND statement to eliminate any BIND scoping problems. The %ProcedureInitialize #EMBED statement is the first programmer-available embed point in the executable code portion of the procedure.

The next six Clarion language statements are directly generated into the procedure to set it up for the action it should perform, as signaled to the procedure through the GlobalRequest variable. The %ProcedureSetup #EMBED statement is the next programmer-available embed point in the executable code portion of the procedure.

The IF KEYCODE() = MouseRight structure detects when the procedure has been called as a result of a RIGHT-CLICK popup menu. If so, it ensures that the keycode is cleared to prevent multiple execution.

The %StandardFormula #INSERT statement generates all the "Procedure Setup" class formulas. Following that, the #INSERT(%FileControlOpen) generates the code to open all the files used in the procedure (if they are not already open). This #GROUP also contains two #EMBED statements that allow the programmer to embed code either before or after the files are opened.

The #INSERT(%StandardWindowOpening) generates the OPEN(window) statement, and the .INI file handling code (if the programmer has checked the **Use .INI file to save and restore program settings** box). This #GROUP also includes two #EMBED statements that allow the programmer to embed code either before or after opening the window.

The next two #EMBED statements allow the programmer to embed code before entering the procedure's ACCEPT loop. #MESSAGE displays its message during source generation.

The ACCEPT loop is Clarion's event handling structure. The next #EMBED (%AcceptLoopBeforeEventHandling) allows the programmer to add code that will be the first to "see" any event that ACCEPT passes on. The CASE EVENT() structure contains all the code to handle field-independent events, generated from the #INSERT(%StandardWindowHandling) statement. This #GROUP is discussed in detail in its own section. The two #EMBED statements that surround this #INSERT and the one following the CASE EVENT structure all give the programmer the opportunity to explicitly handle anyt field-independent event not covered by the generated code.

The #SUSPEND statement means that conditional code statements (those prefaced with #?) will only generate if an explicit code statement (without #?) is also generated for the event, or if the programmer has placed some embedded source or used a Code template in an embed point associated with the event being processed. This is the mechanism that allows Clarion's Template Language to only generate code that is actually required for the procedure, eliminating unnecessary generated code.

The #?CASE ACCEPTED() structure contains all the code to handle all the Accepted events for menu items. Since menu items only generate Accepted events, this structure keeps the following CASE FIELD() structure from becoming unwieldy. This line of code, since it is prefaced with #?, will only generate if there is some other code generated within it, eliminating an empty CASE structure. The code for the CASE structure is generated by the #INSERT(%StandardAcceptedHandling) statement. This #GROUP is also discussed in detail in its own section. The #?END statement will only generate an END statement if other code has already been generated.

The #RESUME statement terminates the #SUSPEND section. If no source code has actually been generated, none of the conditional source statements (prefaced by #?) between the #SUSPEND and the #RESUME generate.

The #?CASE FIELD() structure (also bracketed within #SUSPEND and #RESUME statements) contains all the code to handle all the field-specific events. The code for the CASE structure is generated by the #INSERT(%StandardControlHandling) statement (between its two #EMBED statements). This #GROUP is also discussed in detail in its own section. The #?END statement will only generate an END statement if other code has already been generated. The #EMBED immediately following #RESUME provides an embed point at the bottom of the ACCEPT loop.

The END statement terminates the ACCEPT loop. This statement is always generated (as is the ACCEPT) because every window requires an ACCEPT loop directly associated with it to process the events for that window. The DO ProcedureReturn statement calls the "cleanup code" for the procedure.

The ProcedureReturn ROUTINE begins with a comment block that generates into the Clarion code to explain the ROUTINE's purpose. The first line of code in the ROUTINE is the #INSERT(%FileControlClose) statement. This generates the code to close the files that were opened by the procedure. This #GROUP also contains two #EMBED statements that allow the programmer to embed code either before or after the files are closed.

The #INSERT(%StandardWindowClosing) generates the CLOSE(*window*) statement , and the .INI file handling code (if the programmer has checked the **Use .INI file to save and restore program settings** box). This #GROUP also includes two #EMBED statements that allow the programmer to embed code either before or after closing the window.

The next #EMBED allows the programmer to embed code before closing the procedure's window. The next #INSERT statement generates all the "Procedure Exit" class formulas. The next five Clarion language statements set up the procedure to alert the calling procedure to the action it performed, signaled back to the calling procedure through the GlobalRequest variable. The POPBIND statement eliminates any BIND scoping problems. The #IF structure then determines whether the procedure returns a value and generates the correct RETURN statement.

The InitializeWindow ROUTINE is a standard routine in all of Clarion's shipping Templates. It starts with a comment block, then the #EMBED allows the programmer to perform any initialization code for themselves, and provides Code, Control ,and Extension templates a place to generate their window initialization code. The DO RefreshWindow statement calls the routine to display the current contents of all the controls' USE variables at the time the window is initialized.

The RefreshWindow ROUTINE is another standard routine in all of Clarion's shipping Templates that performs the procedure's MANY:1 lookups and refreshes the screen to ensure any changed data correctly displays to the user at all times. The ROUTINE starts with a comment block then the IF %Window{PROP:AcceptAll} THEN EXIT. statement. This detects when the procedure is on "non-stop" mode performing all data validity checks prior to writing a record to disk, and aborts the re-display.

The first #EMBED allows the programmer to embed code before the lookups. The next #INSERT generates all the "Before Lookups" class formulas, then #INSERT(%StandardSecondaryLookups) generates the code to get all the related records for the procedure. The next #INSERT generates all the "After Lookups" class formulas, then comes a #EMBED to allow the programmer to embed code after the lookups. The DISPLAY statement puts any changed values on screen, and ForceRefresh = False turns off the procedure's screen refresh flag.

The SyncWindow ROUTINE is also a standard routine in all of Clarion's shipping Templates. It performs the same lookups as the RefreshWindow ROUTINE, with similar embed points, but does not refresh the screen. Instead, it ensures all record buffers contain correct data. This ROUTINE is usually called before executing some action that may require the currently highlighted record in a LIST.

The next to last #EMBED statement allows the programmer to embed any ROUTINEs they have called from their code within other embed points. The last #EMBED statement allows any other templates to embed any local PROCEDUREs that are called from their code.

%StandardWindowHandling #GROUP

This #GROUP generates all the code to handle field-independent events for the procedure. It generates its code inside the Window template's CASE EVENT() structure.

```
#GROUP(%StandardWindowHandling)
#FOR(%WindowEvent)
  #SUSPEND
#?OF EVENT:%WindowEvent
  #EMBED(%WindowEventHandling,'Window Event Handling'),%WindowEvent
  #CASE(%WindowEvent)
  #OF('OpenWindow')
  IF NOT WindowInitialized
    DO InitializeWindow
    WindowInitialized = True
  END
    #IF(%FirstField)
  SELECT(%FirstField)
    #ENDIF
  #OF('GainFocus')
  ForceRefresh = True
  IF NOT WindowInitialized
    DO InitializeWindow
    WindowInitialized = True
  ELSE
    DO RefreshWindow
  END
  #OF('Sized')
  ForceRefresh = True
  DO RefreshWindow
  #ENDCASE
  #EMBED(%PostWindowEventHandling,'Window Event Handling, after generated code') %|
         ,%WindowEvent
  #RESUME
#ENDFOR
OF Event:Rejected
  #EMBED(%WindowEventHandlingBeforeRejected,'Window Event Handling - Before Rejected')
  BEEP
  DISPLAY(?)
  SELECT(?)
  #EMBED(%WindowEventHandlingAfterRejected,'Window Event Handling - After Rejected')
#SUSPEND
#?ELSE
  #EMBED(%WindowOtherEventHandling,'Other Window Event Handling')
#RESUME
```

This #GROUP starts with #FOR(%WindowEvent). This means it will loop through every instance of the %WindowEvent symbol, generating code (if required) for each field-independent event in the procedure.

The #SUSPEND statement begins the section of code that will only conditionally generate code if an explicit code statement (without #?) generates, or if the programmer has placed some embedded source or used a Code or Extension template to generate code into an embed point.

The #?OF EVENT:%WindowEvent statement conditionally generates an OF clause to the CASE EVENT() structure for the currently processing instance of %WindowEvent. This line of code, since it is prefaced with #?, will only generate if there is some other code generated within it, eliminating an empty OF clause.

The #EMBED statement is the key to the source generation process, and to the Procedure template's interaction with Code, Control, and Extension templates. Because it has the ",%WindowEvent" appended to the end, the programmer will have a separate embed point available for every instance of the %WindowEvent symbol. This means programmers can write their own code for any field-independent event . It also means any Code, Ccontrol, or Eextension templates the programmer places in the procedure can generate code into these embed points, as needed, to produce the code necessary to support their functionality. These embed points are the targets of the #AT statements used in the Code, Control, and Extension templates.

The #CASE(%WindowEvent) structure generates explicit source code for the field-independent events in its structure. The #OF('OpenWindow') checks for EVENT:OpenWindow and generates the check on the WindowInitialized variable to conditionally initialize the windowand set WindowInitialized to true. This code executes if no EVENT:GainFocus has already occurred (such as opening a second window on the same execution thread that currently has focus). The SELECT(%FirstField) statement is generated only if there are any controls that can receive focus in the window.

The #OF('GainFocus') statement checks for EVENT:GainFocus and generates the ForceRefresh = True, then checks to see if the window has already been initialized (if the user is switching between active threads it would have been). If not, it initializes the window, otherwise it simply refreshes it. The #OF('Sized') statement checks for EVENT:Sized and generates ForceRefresh = True and DO RefreshWindow to refresh the window after the user has resized it.

The #ENDCASE statement terminates the #CASE structure. The next #EMBED allows the programmer an opportunity to embed their own code following the generated code for any of these events. The #RESUME statement terminates the #SUSPEND section. If no source code has actually been generated, no conditional source statements (prefaced by #?) between the #SUSPEND and the #RESUME are generated.

#ENDFOR terminates the #FOR loop, then OF EVENT:Rejected generates and offers before and after embed points surrounding standard code to alert the user that the data input into the current control has been rejected (usually the data is out of range) and leave them on the same control.

The #SUSPEND statement begins another conditional generation section. This means the #?ELSE statement only generates an ELSE if source code is generated by the #EMBED statement. #RESUME terminates this #SUSPEND section.

%StandardAcceptedHandling #GROUP

This #GROUP generates all the code to handle field-specific events for the procedure. It generates its code inside the Window Template's CASE FIELD() structure.

```
#GROUP(%StandardAcceptedHandling)
#FOR(%Control),WHERE(%ControlMenu)
  #FIX(%ControlEvent,'Accepted')
  #MESSAGE('Control Handling: ' & %Control,3)
  #SUSPEND
#?OF %Control
  #EMBED(%ControlPreEventHandling,'Control Event Handling, before generated code') %|
 ,%Control,%ControlEvent
  #INSERT(%FieldTemplateStandardHandling)
  #EMBED(%ControlEventHandling,'Internal Control Event Handling')  %|
        ,%Control,%ControlEvent,HIDE
  #EMBED(%ControlPostEventHandling,'Control Event Handling, after generated code') %|
        ,%Control,%ControlEvent
  #RESUME
#ENDFOR
```

This code starts with the #FOR(%Control),WHERE(%ControlMenu) statement. The WHERE attribute limits this #FOR loop to only those instances of %Control that contain menu items. The #FIX statement ensure that this code only deals with Accepted events.

The #MESSAGE statement displays its message during source generation. #SUSPEND begins a conditional source generation section.

The #?OF %Control statement conditionally generates an OF clause to the CASE ACCEPTED() structure for the currently processing instance of %Control. This line of code, since it is prefaced with #?, will only generate if there is some other code generated within it, eliminating an empty OF clause.

All three #EMBED statements have ",%Control,%ControlEvent" appended to the end, so the programmer will have a separate embed point available for every instance of the %ControlEvent symbol within every instance of the %Control symbol. For this group, this only means the Accepted event.

The #INSERT(%FieldTemplateStandardHandling) statement generates code to handle all the Actions dialog selections the programmer has made for the menu item. The next two #EMBED statements also have ",%Control,%ControlEvent" appended to the end. The first has the HIDE attribute, so it is available only for Code, Control, and Extension template use. These three #EMBEDs give the programmer an embed point both before and after any code automatically generated for them by the Actions tab prompts.

#RESUME terminates this #SUSPEND section. #ENDFOR terminates the %Control loop.

%StandardControlHandling #GROUP

This #GROUP generates all the code to handle field-specific events for the procedure. It generates its code inside the Window template's CASE FIELD() structure.

```
#GROUP(%StandardControlHandling)
#FOR(%Control),WHERE(%Control)
  #MESSAGE('Control Handling: ' & %Control,3)
  #SUSPEND
#?OF %Control
  #EMBED(%ControlPreEventCaseHandling,'Control Handling, before event handling') %|
        ,%Control
  #?CASE EVENT()
    #IF(NOT %ControlMenu)
      #FOR(%ControlEvent)
        #SUSPEND
  #?OF EVENT:%ControlEvent
    #EMBED(%ControlPreEventHandling,'Control Event Handling, Before Generated Code') %|
          ,%Control,%ControlEvent
    #INSERT(%FieldTemplateStandardHandling)
    #EMBED(%ControlEventHandling,'Internal Control Event Handling') %|
          ,%Control,%ControlEvent,HIDE
    #EMBED(%ControlPostEventHandling,'Control Event Handling, After Generated Code') %|
          ,%Control,%ControlEvent
        #RESUME
      #ENDFOR
    #ELSE
  #?OF EVENT:Accepted
    #ENDIF
    #SUSPEND
  #?ELSE
    #EMBED(%ControlOtherEventHandling,'Other Control Event Handling'),%Control
  #RESUME
  #?END
  #EMBED(%ControlPostEventCaseHandling,'Control Handling, after event handling') %|
        ,%Control
  #RESUME
#ENDFOR
```

This code starts with the #FOR(%Control),WHERE(%Control) statement. The WHERE clause may at first seem redundant, since #FOR will only loop through existing instances of %Control. However, since some controls do not need (and so do not have) field equate labels, there are valid instances of %Control that do not contain a value for %Control itself. Therefore, the WHERE attribute limits this #FOR loop to those instances of %Control that do contain a field equate label for the control.

The #MESSAGE statement displays its message during source generation. #SUSPEND begins a conditional source generation section.

The #?OF %Control statement conditionally generates an OF clause to the CASE FIELD() structure for the currently processing instance of %Control. This line of code, since it is prefaced with #?, will only generate if there is some other code generated within it, eliminating an empty OF clause.

The first #EMBED allows the programmer to handle any situation that needs to be handled before any generated code for the control. The #?CASE EVENT() conditionally generates a CASE EVENT() structure for the control. The #IF(NOT %ControlMenu) statement filters out all the menu items, since they are handled by the %StandardAcceptedHandling #GROUP. #FOR(%ControlEvent) loops through all the possible events that the control being processed can generate.

#SUSPEND begins another conditional source generation section, nested within the previous one. This allows multiple levels of conditional source code generation. The outer section is automatically generated if any code is generated from the inner section.

The #?OF EVENT:%ControlEvent statement conditionally generates an OF clause to the CASE EVENT() structure for the currently processing instance of %Control. This line of code, since it is prefaced with #?, will only generate if there is some other code generated within it, eliminating an empty OF clause.

This next #EMBED statement has ",%Control,%ControlEvent" appended to the end, so the programmer will have a separate embed point available for every instance of the %ControlEvent symbol within every instance of the %Control symbol. This means programmers can write their own code for any field-specific event, for any control . It also means any Code templates, Control templates, or Extension templates the programmer places in the procedure can generate code into these embed points, as needed, to produce the code necessary to support their functionality. These embed points are the targets of the #AT statements used in the Code, Control, and Extension templates.

The #INSERT(%FieldTemplateStandardHandling) statement generates code to handle all the Actions tab selections the programmer has made for the control. The prompts on the Actions tab come from the #FIELD structures that were #INSERTed at the beginning of the Window template.

The next two #EMBED statements also have ",%Control,%ControlEvent" appended to the end, so the programmer will have a separate embed point available for every instance of the %ControlEvent symbol within every instance of the %Control symbol. The first has the HIDE attribute, so it is available only for Code, Control, and Extension template use. These three #EMBEDs give the programmer an embed point both before and after any code automatically generated for them by the Actions tab prompts.

#RESUME terminates the inner conditional source generation section, then #ENDFOR terminates the %ControlEvent loop. The #ELSE refers back to the #IF(NOT %ControlMenu) and will generate an empty OF EVENT:Accepted followed by an ELSE statement for a Menu item if the programmer has entered code into the Other Control Event Handling embed point. This eliminates any duplication between EVENT:Accepted code for a menu item while still allowing the programmer to process any user-defined events for them.

The #SUSPEND statement begins another nested conditional generation section. This means the #?ELSE statement only generates an ELSE if source code is generated by the #EMBED statement. #RESUME terminates this #SUSPEND section.

The #?END generates the END statement for the CASE FIELD() structure, if any code has been generated, then the #RESUME statement terminates the outer #SUSPEND section. #ENDFOR terminates the %Control loop.

Code Template: ControlValueValidation

The *ControlValueValidation* Code template performs data entry validation for an entry-type control (ENTRY, SPIN, or COMBO) by looking up the value entered by the user in another data file. If the lookup is successful, the entered value is valid. If not, it calls another procedure to allow the user to select a valid value from the lookup file. This Code template is designed to generate code only into EVENT:Selected or EVENT:Accepted embed points of an ENTRY, SPIN, or COMBO control. These are the controls into which a user can directly type in data.

```
#CODE(ControlValueValidation,'Control Value Validation')
  #RESTRICT
    #CASE(%ControlType)
    #OF('ENTRY')
    #OROF('SPIN')
    #OROF('COMBO')
      #CASE(%ControlEvent)
      #OF('Accepted')
      #OROF('Selected')
        #ACCEPT
      #ELSE
        #REJECT
      #ENDCASE
    #ELSE
      #REJECT
    #ENDCASE
  #ENDRESTRICT
#DISPLAY('This Code Template is used to perform a control value')
#DISPLAY('validation.  This Code Template only works for')
#DISPLAY('the Selected or Accepted Events for an Entry Control.')
#DISPLAY('')
#PROMPT('Lookup Key',KEY),%LookupKey,REQ
#PROMPT('Lookup Field',COMPONENT),%LookupField,REQ
#PROMPT('Lookup Procedure',PROCEDURE),%LookupProcedure
#DISPLAY('')
#DISPLAY('The Lookup Key is the key used to perform the value validation.')
#DISPLAY('If the Lookup Key is a multi-component key, you must insure that')
#DISPLAY('other key elements are primed BEFORE this Code Template is used.')
#DISPLAY('')
#DISPLAY('The Lookup field must be a component of the Lookup Key.  Before')
#DISPLAY('execution of the lookup code, this field will be assigned the value of')
#DISPLAY('the control being validated, and the control will be assigned the value')
#DISPLAY('of the lookup field if the Lookup procedure is successful.')
#DISPLAY('')
#DISPLAY('The Lookup Procedure is called to let the user to select a value. ')
#DISPLAY('Request upon entrance to the Lookup will be set to SelectRecord, and ')
#DISPLAY('successful completion is signalled when Response = RequestCompleted.')
#IF(%ControlEvent='Accepted')
```

```
 IF %Control{PROP:Req} = False AND NOT %ControlUse #<! If not required and empty
ELSE
 #INSERT(%CodeTPLValidationCode)
END
#ELSIF(%ControlEvent='Selected')
 #INSERT(%CodeTPLValidationCode)
#ELSE
  #ERROR('This Code Template must be used for Accepted or Selected Events!')
#ENDIF
```

A Code template always starts with the #CODE statement, which identifies it within the template set and defines the description which appears in the list of available Code templates for a given embed point.

The #RESTRICT structure defines the embed points where the code template will appear as a choice. The #CASE(%ControlType) structure limits the embed points to the ENTRY, SPIN, and COMBO controls, and the #CASE(%ControlEvent) structure limits the embed points to EVENT:Accepted and EVENT:Selected.

The #ACCEPT statement indicates these are appropriate embed points, while the #REJECT indicates all other control type and event embed points are not valid for the Code template to appear in as a choice.

All the #DISPLAY statements display their text to the programmer on the code template's prompt dialog. These describe the information the programmer needs to supply in the prompts.

The first #PROMPT asks for the name of the key to use in the file that will be used to validate the user's input. The REQ attribute indicates the programmer must supply this information.

The second #PROMPT asks for the name of the field in the key that contains the same information the user should enter into the control. Again, the REQ attribute indicates the programmer must supply this information.

The third #PROMPT asks for the name of the procedure to call if the lookup is unsuccessful. This would usually be a Browse procedure for the lookup file with a Select button to allow the user to choose the record containing the value they want for the control.

Again, the #DISPLAY statements display text to the programmer on the prompt dialog to describe the information the programmer needs to supply in the prompts.

The #IF(%ControlEvent='Accepted') structure generates an IF structure for EVENT:Accepted that detects when the control has the REQ attribute or the user has entered a value and #INSERTs the %CodeTPLValidationCode #GROUP to generate the source code for the data validation. The #ELSIF just unconditionally #INSERTs the %CodeTPLValidationCode #GROUP to generate the source code for the data validation.

If the event is anything other than EVENT:Accepted or EVENT:Selected, an error message is the only output generated.

%CodeTPLValidationCode #GROUP

This #GROUP is the "workhorse" of the Code template. It generates the actual file lookup code to validate the data entry. It takes the information provided in the prompts and combines it with the %ControlUse symbol to generate a GET statement into the lookup file. If the GET is successful, the data is valid. If not, it calls the lookup procedure.

```
#GROUP(%CodeTPLValidationCode)
 %LookupField = %ControlUse                      #<! Move value for lookup
#FIND(%Field,%LookupField)                       #! FIX field for lookup
 GET(%File,%LookupKey)                           #<! Get value from file
 IF ERRORCODE()                                  #<! IF record not found
   GlobalRequest = SelectRecord                  #<! Set Action for Lookup
   %LookupProcedure                              #<! Call Lookup Procedure
   LocalResponse = GlobalResponse                #<! Save Returned Action
   GlobalResponse = RequestCancelled             #<! Clear the Action Value
   IF LocalResponse = RequestCompleted           #<! IF Lookup successful
     %ControlUse = %LookupField                  #<! Move value to control field
#IF(%ControlEvent='Accepted')                    #! IF a Post-Edit Validation
   ELSE                                          #<! ELSE (IF Lookup NOT...)
     SELECT(%Control)                            #<! Select the control
     CYCLE                                       #<! Go back to ACCEPT
#ENDIF                                           #! END (IF a Pre-Edit...)
   END                                           #<! END (IF Lookup successful)
#IF(%ControlEvent='Selected')                    #! IF a Pre-Edit Validation
   SELECT(%Control)                              #<! Select the control
#ENDIF                                           #! END (IF a Pre-Edit...)
 END                                             #<! END (IF record not found)
```

This #GROUP starts by generating the %LookupField = %ControlUse assignment. This assigns the control's USE variable to the field named in the second prompt; the key field that should contain the correct value.

The #FIND(%Field,%LookupField) statement looks through all the fields in the data dictionary, looking for a matching field to the one contained in %LookupField. This fixes %Field and %File to the correct values to generate the GET(%File,%LookupKey) statement. This becomes a GET(file,key) form of the GET statement to get a single record from the lookup file with matching key field values.

The IF ERRORCODE() structure checks for a successful GET operation. If an error occurred, the GET was unsuccessful and the GlobalRequest = SelectRecord statement sets up the call to the lookup procedure, generated by the %LookupProcedure statement.

After return from the lookup procedure, LocalResponse = GlobalResponse saves the lookup procedure's response code. Then the GlobalResponse = RequestCancelled statement cleans up so any other execution thread does not get an incorrect response. This must be done immediately, before the user has a chance to change execution threads.

The IF LocalResponse = RequestCompleted structure detects a user choice from the lookup procedure and the %ControlUse = %LookupField statement assigns the choice to the control's USE variable.

The #IF(%ControlEvent='Accepted') detects when the Code template is generating for EVENT:Accepted and adds the ELSE clause to SELECT the control and CYCLE back to the top of the ACCEPT loop.

The END statement terminates the IF LocalResponse = RequestCompleted structure. The #IF(%ControlEvent='Selected') structure generates the SELECT statement for the control when generating for EVENT:Selected.

The END statement terminates the IF ERRORCODE() structure. Obviously, if there was no error on the GET statement, the data is valid and no further code is necessary.

Control Template: DOSFileLookup

The *DOSFileLookup* Control template adds an ellipsis (...) button which leads the end user to a standard **Open File** dialog. You can specify a file mask, and a return variable to hold the end user's choice.

```
#CONTROL(DOSFileLookup,'Lookup a DOS file name'),WINDOW,MULTI
  CONTROLS
    BUTTON('...'),AT(,,12,12),USE(?LookupFile)
  END
#BOXED('DOS File Lookup Prompts')
  #PROMPT('&File Dialog Header:',@S60),%DOSFileDialogHeader,REQ,DEFAULT('Choose File')
  #PROMPT('&DOS FileName Variable:',FIELD),%DOSFileField,REQ
  #PROMPT('D&efault Directory:',@S80),%DOSInitialDirectory
  #PROMPT('&Return to original directory when done.',CHECK),%ReturnToOriginalDir,AT(10)
  #PROMPT('&Use a variable to specify the file mask(s).',CHECK),%DOSVariableMask,AT(10)
  #ENABLE(%DOSVariableMask)
    #PROMPT('Vari&able Mask Value:',FIELD),%DOSVariableMaskValue
  #ENDENABLE
  #ENABLE(NOT %DOSVariableMask)
    #PROMPT('F&ile Mask Description:',@S40),%DOSMaskDesc,REQ,DEFAULT('All Files')
    #PROMPT('Fi&le Mask',@S50),%DOSMask,REQ,DEFAULT('*.*')
    #BUTTON('More Fil&e Masks'),MULTI(%DOSMoreMasks,%DOSMoreMaskDesc&'-'&%DOSMoreMask)
      #PROMPT('File Mask Description:',@S40),%DOSMoreMaskDesc,REQ
      #PROMPT('File Mask',@S50),%DOSMoreMask,REQ
    #ENDBUTTON
  #ENDENABLE
#ENDBOXED
#LOCALDATA
DOSDialogHeader      CSTRING(40)
DOSExtParameter      CSTRING(250)
DOSTargetVariable    CSTRING(80)
#ENDLOCALDATA
#ATSTART
  #DECLARE(%DOSExtensionParameter)
  #DECLARE(%DOSLookupControl)
  #FOR(%Control),WHERE(%ControlInstance = %ActiveTemplateInstance)
    #SET(%DOSLookupControl,%Control)
  #ENDFOR
  #IF(NOT %DOSVariableMask)
    #SET(%DOSExtensionParameter,%DOSMaskDesc & '|' & %DOSMask)
    #FOR(%DOSMoreMasks)
      #SET(%DOSExtensionParameter,%DOSExtensionParameter & '|' & %DOSMoreMaskDesc &|
           & '|' & %DOSMoreMask)
    #ENDFOR
  #END
#ENDAT
```

```
#AT(%ControlEventHandling,%DOSLookupControl,'Accepted')
IF NOT %DOSFileField
  #INSERT(%StandardValueAssignment,'DOSTargetVariable',%DOSInitialDirectory)
ELSE
  DOSTargetVariable = %DOSFileField
END
#INSERT(%StandardValueAssignment,'DOSDialogHeader',%DOSFileDialogHeader)
#IF(%DOSVariableMask)
DOSExtParameter = %DOSVariableMaskValue
#ELSE
DOSExtParameter = '%DOSExtensionParameter'
#ENDIF
#IF(%ReturnToOriginalDir)
IF FILEDIALOG(DOSDialogHeader,DOSTargetVariable,DOSExtParameter,FILE:KeepDIR)
#ELSE
IF FILEDIALOG(DOSDialogHeader,DOSTargetVariable,DOSExtParameter,0)
#ENDIF
  %DOSFileField = DOSTargetVariable
  DO RefreshWindow
END
#ENDAT
```

This starts, as all Control templates must, with a #CONTROL statement. The WINDOW attribute allows you to populate it onto a window, but not onto a report. The MULTI attribute specifies that the template may be populated multiple times ontothe same window. The CONTROLS section pre-defines the BUTTON control for the window.

The #BOXED structure places a box around all the prompts that display on the Actions tab for this Control template. The first #PROMPT asks for the text for the caption of the **Open File** dialog, and the next asks for the name of a variable to receive the end user's choice. The third allows you to explicitly set the directory in which the **Open File** dialog starts.

The fourth #PROMPT is a check box asking whether the program should return to the directory from which it started from the **Open File** dialog. The fifth #PROMPT is a check box asking whether the programmer will explicitly set the file mask(s) for the **Open File** dialog, or use a variable to determine them at run time. When checked, the first #ENABLE activates the Variable Mask Value #PROMPT to get the name of the variable toi use at run time. If not checked, the second #ENABLE activates its set of prompts to get each explicit file mask to pass to the **Open File** dialog.

The #LOCALDATA section defines three local variables that generate automatically as part of the procedure. These local variables are only used in the code generated by this Control template as the actual variables passed as parameters to the FILEDIALOG procedure.

The #ATSTART statement begins a section of template code that executes before any source code generates for the procedure. This means it is only appropriate to initialize user-defined template symbols and perform any necessary set up to generate correct source for the control template into the procedure. This section does not generate source code. The #DECLARE statements declare two symbols used only during source generation for this Control template.

#FOR(%Control),WHERE(%ControlInstance=%ActiveTemplateInstance) executes the enclosed #SET statement only for the single control populated by this Control template. The #SET statement then places the field equate label of the control into %DOSLookupControl.

The #IF structure checks whether the programmer checked the **Use a Variable to specify the file mask(s)** box and if not, sets up the specific file masks the programmer chose to pass to the FILEDIALOG procedure. The #ENDAT statement terminates the #ATSTART section.

The next #AT generates Clarion code into the embed point for the Accepted event for the control populated by this Control template to perform the file lookup. The IF NOT %DOSFileField structure detects whether the user has performed the lookup. If they haven't the initial directory is assigned to the DOSTargetVariable. If the user has performed the lookup, the ELSE clause assigns the result of the previous lookup as the starting point for the next.

The #INSERT statement creates assignment statements to initialize the **File Open** dialog's title. Next, the #IF(%DOSVariableMask) conditionally generates an assignment to initialize the file mask to pass to the **File Open** dialog.

The #IF (%ReturnToOrigianlDir) generates the correct IF FILEDIALOG structure performs the actual lookup for the file, either to return to the original directory or not. If the user selects a file from the **File Open** dialog, the selected filename is assigned to variable the user selected in the **DOS FileName Variable** prompt, then he DO RefreshWindow statement ensures that all data on the window is current. The #ENDAT statement terminates the #AT section.

Extension Template: DateTimeDisplay

The *DateTimeDisplay* Extension template displays the date and/or time in either a display-only STRING control or a section of the status bar. Of course, the status bar should be declared on the window.

```
#EXTENSION(DateTimeDisplay,'Display the date and/or time in the current window') %|
          ,HLP('~TPLExtensionDateTimeDisplay'),PROCEDURE
#BUTTON('Date and Time Display'),AT(10,,180)
  #BOXED('Date Display...')
    #PROMPT('Display the current day/date in the window',CHECK) %|
           ,%DisplayDate,DEFAULT(0),AT(10,,150)
    #ENABLE(%DisplayDate)
      #PROMPT('Date Picture:',DROP('October 31, 1959|OCT 31,1959|10/31/59| %|
                                11/31/1959|31 OCT 59|31 OCT 1959|31/10/59| %|
                                          31/10/1959|Other')),%DatePicture %|
                                   ,DEFAULT('October 31, 1959')
      #ENABLE(%DatePicture = 'Other')
        #PROMPT('Other Date Picture:',@S20),%OtherDatePicture,REQ
      #ENDENABLE
      #PROMPT('Show the day of the week before the date',CHECK),%ShowDayOfWeek %|
              ,DEFAULT(1),AT(10,,150)
      #PROMPT('&Location of Date Display:',DROP('Control|Status Bar')) %|
              ,%DateDisplayLocation
      #ENABLE(%DateDisplayLocation='Status Bar')
        #PROMPT('Status Bar Section:',@n1),%DateStatusSection,REQ,DEFAULT(1)
      #ENDENABLE
      #ENABLE(%DateDisplayLocation='Control')
        #PROMPT('Date Display Control:',CONTROL),%DateControl,REQ
      #ENDENABLE
    #ENDENABLE
  #ENDBOXED
  #BOXED('Time Display...')
    #PROMPT('Display the current time in the window',CHECK),%DisplayTime %|
            ,DEFAULT(0),AT(10,,150)
    #ENABLE(%DisplayTime)
      #PROMPT('Time Picture:',DROP('5:30PM|5:30:00PM|17:30|17:30:00| %|
                                   1730|173000|Other')),%TimePicture %|
                                   ,DEFAULT('5:30PM')
      #ENABLE(%TimePicture = 'Other')
        #PROMPT('Other Time Picture:',@S20),%OtherTimePicture,REQ
      #ENDENABLE
      #PROMPT('&Location of Time Display:',DROP('Control|Status Bar')) %|
              ,%TimeDisplayLocation
      #ENABLE(%TimeDisplayLocation='Status Bar')
        #PROMPT('Status Bar Section:',@n1),%TimeStatusSection,REQ,DEFAULT(2)
      #ENDENABLE
      #ENABLE(%TimeDisplayLocation='Control')
```

```
        #PROMPT('Time Display Control:',CONTROL),%TimeControl,REQ
      #ENDENABLE
    #ENDENABLE
  #ENDBOXED
#ENDBUTTON
#ATSTART
  #DECLARE(%TimerEventGenerated)
  #IF(%DisplayDate)
    #DECLARE(%DateUsePicture)
    #CASE(%DatePicture)
    #OF('10/31/59')
      #SET(%DateUsePicture,'@D1')
    #OF('10/31/1959')
      #SET(%DateUsePicture,'@D2')
    #OF('OCT 31,1959')
      #SET(%DateUsePicture,'@D3')
    #OF('October 31, 1959')
      #SET(%DateUsePicture,'@D4')
    #OF('31/10/59')
      #SET(%DateUsePicture,'@D5')
    #OF('31/10/1959')
      #SET(%DateUsePicture,'@D6')
    #OF('31 OCT 59')
      #SET(%DateUsePicture,'@D7')
    #OF('31 OCT 1959')
      #SET(%DateUsePicture,'@D8')
    #OF('Other')
      #SET(%DateUsePicture,%OtherDatePicture)
    #ENDCASE
  #ENDIF
  #IF(%DisplayTime)
    #DECLARE(%TimeUsePicture)
    #CASE(%TimePicture)
    #OF('17:30')
      #SET(%TimeUsePicture,'@T1')
    #OF('1730')
      #SET(%TimeUsePicture,'@T2')
    #OF('5:30PM')
      #SET(%TimeUsePicture,'@T3')
    #OF('17:30:00')
      #SET(%TimeUsePicture,'@T4')
    #OF('173000')
      #SET(%TimeUsePicture,'@T5')
    #OF('5:30:00PM')
      #SET(%TimeUsePicture,'@T6')
    #OF('Other')
      #SET(%TimeUsePicture,%OtherTimePicture)
    #ENDCASE
  #ENDIF
#ENDAT
#AT(%DataSectionBeforeWindow)
  #IF(%DisplayDate AND %ShowDayOfWeek)
DisplayDayString STRING('Sunday   Monday   Tuesday  WednesdayThursday %|
                         Friday   Saturday ')
DisplayDayText   STRING(9),DIM(7),OVER(DisplayDayString)
  #ENDIF
#ENDAT
#AT(%BeforeAccept)
  #IF(%DisplayTime OR %DisplayDate)
IF NOT INRANGE(%Window{Prop:Timer},1,100)
  %Window{Prop:Timer} = 100
END
#INSERT(%DateTimeDisplayCode)
  #ENDIF
#ENDAT
#AT(%WindowEventHandling,'Timer')
  #SET(%TimerEventGenerated,%True)
  #IF(%DisplayDate OR %DisplayTime)
#INSERT(%DateTimeDisplayCode)
  #ENDIF
#ENDAT
#AT(%WindowOtherEventHandling)
  #IF(%DisplayDate OR %DisplayTime)
    #IF(NOT %TimerEventGenerated)
IF EVENT() = EVENT:Timer
  #INSERT(%DateTimeDisplayCode)
END
    #ENDIF
  #ENDIF
#ENDAT
```

An Extension template starts with the #EXTENSION statement. The PROCEDURE attribute specifies the Extension template is available only at the procedure level, not the global level of the application.

The #BUTTON structure creates a separate page for all the prompts for this Extension template. These prompts ask the programmer for the format of the date and/or time to display, and whether to display them in a control or the status bar.

The #ATSTART statement begins a section of template code that executes before any source code generates for the procedure. This means it is only appropriate to initialize user-defined template symbols and perfrom any necessary set up to generate correct source for the control template into the procedure. This section does not generate source code.

The #DECLARE(%TimerEventGenerated) statement declares a symbol used only in this Extension template. It is used to flag whether an OF EVENT:Timer clause has been generated for the procedure.

The #IF(%DisplayDate) structure sets up to display the date by declaring a symbol to contain the programmer's choice of date formats. The #CASE structure assigns that choice to the %DateUsePicture symbol. The #IF(%DisplayTime) structure sets up to display the time by declaring a symbol to contain the programmer's choice of date formats. The #CASE structure assigns that choice to the %TimeUsePicture symbol. The #ENDAT statement terminates the #ATSTART section.

The next #AT generates code into the embed point that appears immediately before the window data structure. The #IF(%DisplayDate AND %ShowDayOfWeek) structure generates two local variable declarations for the procedure if the programmer is displaying the date with the day of week.

The next #AT generates code into the embed point that appears immediately before the ACCEPT loop. The #IF(%DisplayTime OR %DisplayDate) structure generates code that ensures the window has its TIMER attribute set. The IF NOT INRANGE(%Window{Prop:Timer},1,100) detects the lack of the attribute, then %Window{Prop:Timer} = 100 sets it to one second. The #INSERT(%DateTimeDisplayCode) adds the code that updates the display.

The next #AT generates code into the embed point for EVENT:Timer. This embed point only appears if the programmer has placed a TIMER attribute on the window. Therefore, the #SET(%TimerEventGenerated,%True) statement signals that code was generated in this embed point. The #IF(%DisplayTime OR %DisplayDate) structure ensures the #INSERT(%DateTimeDisplayCode) statement generates the code that updates the display every time EVENT:Timer is processed.

The next #AT generates code into the "Other Window Event Handling" embed point. This is the ELSE clause of the CASE EVENT() structure to handle field-independent events. #IF(NOT %TimerEventGenerated) detects that the previous #AT did not generate code because the programmer did not place the TIMER attribute on the window. Therefore, the IF EVENT() = EVENT:Timer structure is necessary for the code that updates the display whenever EVENT:Timer occurs.

%DateTimeDisplayCode #GROUP

This #GROUP generates the code to actually display the Date and/or Time.

```
#GROUP(%DateTimeDisplayCode)
    #IF(%DisplayDate)
      #IF(%ShowDayOfWeek)
        #CASE(%DateDisplayLocation)
        #OF('Control')
  %DateControl{Prop:Text} = CLIP(DisplayDayText[(TODAY()%%7)+1]) & ', ' &  %|
                                FORMAT(TODAY(),%DateUsePicture)
  DISPLAY(%DateControl)
        #ELSE
  %Window{Prop:StatusText,%DateStatusSection} = CLIP(DisplayDayText[(  %|
          TODAY()%%7)+1]) & ', ' & FORMAT(TODAY(),%DateUsePicture)
        #ENDCASE
      #ELSE
        #CASE(%DateDisplayLocation)
        #OF('Control')
  %DateControl{Prop:Text} = FORMAT(TODAY(),%DateUsePicture)
  DISPLAY(%DateControl)
        #ELSE
  %Window{Prop:StatusText,%DateStatusSection} = FORMAT(TODAY(),%DateUsePicture)
        #ENDCASE
      #ENDIF
    #ENDIF
    #IF(%DisplayTime)
      #CASE(%TimeDisplayLocation)
      #OF('Control')
  %TimeControl{Prop:Text} = FORMAT(CLOCK(),%TimeUsePicture)
  DISPLAY(%DateControl)
      #ELSE
  %Window{Prop:StatusText,%TimeStatusSection} = FORMAT(CLOCK(),%TimeUsePicture)
      #ENDCASE
    #ENDIF
```

The #IF(%DisplayDate) structure generates the code to display the date. The #IF(%ShowDayOfWeek) structure detects the programmer's choice to display the day along with the date, then #CASE(%DateDisplayLocation) generates the code to display into a STRING display-only control for the #OF('Control') clause.

The assignment statement concatenates the day of the week (from the DisplayDayText[(TODAY() %% 7)+1] expression) with the formatted date (from the FORMAT(TODAY(),%DateUsePicture) expression) into the STRING(text) property (the %DateControl{Prop:Text} property). The %% generates as a single % (modulus operator) for the TODAY() % 7 + 1 expression to get the correct day of the week text from the DisplayDayText array. The #ELSE clause of the #CASE(%DateDisplayLocation) assigns the same expression to the status bar section the programmer chose for the date display.

The #ELSE clause of the #IF(%ShowDayOfWeek) structure performs the same assignments, without the day of the week. The #IF(%DisplayTime) structure performs the same type of assignments of the formatted time to either a STRING display-only control or the status bar.

Index: