Exec File Preprocessor

Overview

The exec file preprocessor supports a wide variety of features for generating exec files, including parameterization (with defaults), prompting for input and options, nested exec files, commenting, conditionals, general console I/O, among others. These facilities should allow you to create modular, flexible exec files which are both powerful and easy to maintain.

Essentially the preprocessor provides a language interpreter for the generation of WorkShop commands. The basic operation of the exec preprocessor is as follows. The preprocessor is invoked when the WorkShop shell recognizes a command to run an exec file. The preprocessor then reads and processes its input file(s) and creates a temporary file with the output it generates (which typically consists of WorkShop directives and commands to run other programs). This output file is then passed back to the WorkShop shell for execution. After the execution of the commands in the temporary file has terminated (either having run to completion or having raised an error) the temporary file is deleted by the shell.

Exec file input consists of two types of lines -- **normal lines** with commands which will be passed through to the WorkShop and **exec command lines** which are directed to the preprocessor (and which will not get passed to the WorkShop). Thus, exec files actually are written in two languages -- one directed to the WorkShop and any programs that may be run under the WorkShop, and the other directed to the exec preprocessor itself. So, while the preprocessor has a symbolic, keyword-oriented **meta-language**, the underlying WorkShop command language is the same as usual, i.e., the keystroke-oriented, UCSD-style command interface. That is, the exec file preprocessor provides a high level language for the generation of WorkShop commands, however, it does not provide an alternative to the WorkShop's basic command language.

In the rest of this document the method for constructing and invoking exec files is described. The material is organized in the following sections:

Exec File Invocation	- how to run an exec file	- page 2
The Exec Language	- the exec file command language	- page 3
Examples	- sample exec files	- page 11
Exec Invocation Options	- options for running exec files	- page 15
Exec Programming Tips	- exec file programming hints	- page 19
Exec Errors	- errors reported by preprocessor	- page 20

Exec File Invocation

An invocation line for the preprocessor has the following form:

```
<exec command> <exec file> [ (<parameter list>) [ <exec options> ] ]
```

The <exec command> can be either "EXEC/" or "<". The <exec file> is the name of the exec file you wish to run. A ".TEXT" extension will be assumed if one is not specified; however, you may override the mechanism which supplies the ".TEXT" extension by ending your <exec file> name with a dot; e.g., using "foo." will cause the preprocessor to look for the file "foo" rather than "foo.text".

The optional <parameter list> is enclosed in parentheses. The parameter list may be empty or it may include up to ten parameters delimited by commas. For example, we may have an exec file to run compiles which takes volume and source file parameters, which we might invoke with "compile(foo,-work-)". Parameters may be omitted (leaving them as null paramters) by specifying them with the null string, as in "compile(foo,)", which omits the volume from our previous example. Alternately, parameters may be left unspecified altogether, as in "compile(foo)", in which case they also get null values. One reason for leaving off parameters is that the exec file may have been set up to supply default values, as is described below.

The <exec options> which follow the closing ")" of the parameter list consist of single letter commands which will modify the behavior of the preprocessor; for example, "S" is used to indicate that you want to step through the exec file as it is being processed, conditionally selecting which commands will be sent to the WorkShop shell. The exec options are discussed in detail in the "Exec Invocation Options" section below.

The preprocessor's output is a temporary file with a "..TEXT" extension. The temporary file is the processed version of your exec commands, that is, all preprocessor-oriented commands will have been processed and removed, leaving only the WorkShop-related commands. This temporary file is passed to the WorkShop shell executive when the preprocessor is done. The WorkShop shell will then run the temporary exec file and delete it automatically when completed.

Note that the preprocessor is not case-sensitive, but it does preserve the case of parameters and strings supplied by the user.

The Exec Language

Following is a description of the format of exec files and the language recognized by the preprocessor. Note that the exec language is independent of the underlying command language, that is, the preprocessor recognizes its own commands but does not know anything about the form of the commands generated by the exec file and passed to the system command shell.

The format of exec files is line oriented. The preprocessor recognizes two types of lines -- exec command lines and normal lines -- each of which is processed differently.

Exec Command Lines

Exec command lines are distinguished by having a "$" as their first significant (non-blank) character; they contain commands which control the operation of the preprocessor on the rest of the exec file.

While exec commands have a specific syntax (each command is discussed below), the command may be entered in a "free format", that is, nothing is required to appear in a fixed position in a line as long as the order of the syntactic elements is preserved. Any number of blanks may preceed or follow the "$" of an exec command line; and any number of blanks may appear between the syntactic elements of a command. The case of keywords in the preprocessor language is not significant; however, in this document the keywords appear in uppercase so that they may be more easily recognized.

If an exec command line contains an incomplete command, the preprocessor will automatically go to the next line to continue the command. Note that "$" begins exec commands, and should not appear on the continuation line(s).

Normal Lines

Normal lines contain text which is meant to pass through the preprocessor to the system, that is, normal lines contain system commands and commands directed to programs being run by the exec file.

The only processing that takes place in normal lines is expansion of parameter references, removal of comments, and processing of the literalizing character (see below).

Beginning and Ending an Exec File -- $EXEC and $ENDEXEC

The general form of exec files is they must begin with a "$EXEC" line and must end with a "$ENDEXEC" line. The exceptions to this basic rule (for those miscreants who embed their exec files in their program sources) are: (1) one line of text may preceed the "$EXEC" line if the "I" invocation option is used, and (2) any amount of text may follow the "$ENDEXEC" line, but it will be ignored.

{ Comments }

Comments may appear in both exec command lines and normal lines. Comments are delimited by curly braces. Comments in normal lines are completely removed and do not appear in the generated exec file. Comments may cross line boundries and, in effect, comment out those line boundries.

%n Parameters

Parameters passed to exec files are positional, that is, they are referenced by the index of their position in the parameter list. Up to ten parameters are permitted, numbered 0 through 9. Parameter references are of the form "%n" where n is a single digit.

The values of parameters are strings; they are set either in the parameter list of the exec invocation line or else by one of the preprocessor commands described below. If a "%n" parameter reference appears in a normal line its occurence will be replaced by the string value of that parameter. If a parameter reference appears in an exec command line it will be recognized as a symbolic parameter reference and how it will be treated will depend on the command.

Parameters references are expanded only at the top level; that is, once a parameter reference has been replaced by its corresponding string value, no further expansion will take place in that string (i.e., further parameter references in the string value will not be expanded).

One point to note is that typically not all ten parameters are used, and unusued parameters may be used as temporary variables inside the exec file. (The use of such variables is explained below.)

~ the Literalizing Character

Tilde ("~") may be used as a literalizaing character in normal lines; its effect is to pass the following character through the preprocessor without processing it. This allows you to have a "$" as the first generated character of a normal line without making the line appear to be an exec command line. It also allows you to have a "%" or "{" that will not be interpreted to be the start of a parameter reference or a comment. Tilde itself is represented by "~~".

$SET and $DEFAULT

The SET and DEFAULT commands provide a way of changing the value of a parameter inside of an exec file. The form of these commands is:

```
  $ SET <%n> TO <str expr>
```

and

```
  $ DEFAULT <%n> TO <str expr>
```

where <%n> is a parameter reference and <str expr> is a string expression as described in the following section.

The effect of the SET command is to change the value of the specified parameter to the value of the given string expression. The effect of the DEFAULT command is similar to that of the SET command, however, the assignment only takes place if the value of the specified parameter is the null string when the DEFAULT command is encountered. Thus, this command can be used to supply default values to parameters that have been left unspecified or empty in the exec invocation line.

String Expressions

A string expression (<str expr>) may specify a string by a number of means, as noted in the following grammar.

```
    <str expr>          ::= <parameter reference>
                          | <str constant>
                          | <expanded str constant>
                          | <str function>
                          | <exec function call>
```

A **parameter reference** has the usual "%n" form. A **string constant** has the standard form of text delimited by single quotes ('), with a quote inside the string specified by the double quote rule, as in 'That''s all, folks!'. An **expanded string constant** is similar to a string constant, except that double quotes (") are used as delimiters and parameter references are expanded within the string. A **string function** is a preprocessor function which returns a string value (these are described in the following section). An **exec function call** is an invocation of an exec file which returns a string value (as described in a following section, "Exec Function Calls").

String Functions -- CONCAT and UPPERCASE

The string functions CONCAT and UPPERCASE may be applied to other string expressions to produce new string values.

The **CONCAT** function allows several string expressions to be combined to produce a result which is a single string. The CONCAT function has the form:

```
  CONCAT ( <str expr> [ , <str expr> ]* )
```

That is, CONCAT takes a list of string expressions, separated by commas.

The **UPPERCASE** function converts any lower case letters in its argument to upper case. It has the following form:

```
  UPPERCASE ( <str expr> )
```

An example of the use of this function is

```
  $ SET %0 TO UPPERCASE (%0)
```

which will set parameter 0 to an uppercase version of its previous value.

$REQUEST

The REQUEST command provides a way to prompt for values from the console. Its form is:

```
  $ REQUEST <%n> WITH <str expr>
```

The REQUEST command will print the given string expression to the console and will read a line from the console which it will assign to the specified parameter. Thus the <str expr> is the prompt that you will request with.

$SUBMIT

The SUBMIT command allows nesting of exec files, that is, it allows another exec file to be called from within an exec file. The form of the SUBMIT command is:

```
  $ SUBMIT <exec command>
```

where <exec command> is an exec command of the same form as you would have following the "exec/" or "<" at the WorkShop shell command level. This exec command may include parameter and exec options in the usual fashion.

The effect of the SUBMIT command is to process the specified exec file, putting any generated exec output text into the current exec temporary file. Thus, while a single exec file may have several nested sub-exec files, only a single temporary output file is generated which includes the output generated by all of the input files. Exec files may be nested to an arbitrary level.

Note that only the "I" (Ignore first line) and "B" (Blanks significant) options are valid on a SUBMIT command, while the "R" (ReRun), "S" (Step mode) and "T" (Temporary file saved) options are only applicable from the main exec invocation line.

$RETURN -- Exec Functions

The RETURN command allows exec files to return string values to other (calling) exec files. Thus the RETURN command can turn an exec file into a **function**. The form of the RETURN command is:

```
  $ RETURN [ <str expr> ]
```

Executing a RETURN command will terminate the current exec file and return to the calling exec file with the specified string value. The method by which exec functions are called is described in the following section.

Exec functions can be used to do such things as determining whether a program file (and its corresponding include files, if any) have been modified since their last compilation, and may thus be used to conditionally submit compiles. If written generally enough, such a function could be used by many exec files.

Exec functions can produce side effects, that is, they may contain normal lines which will get placed in the temporary file. While the intentional use of such side effects is unlikely, inadvertent instances may occur and will be potentially hazardous to your exec files. (An unexpected blank line in the middle of an exec file can often throw it out of sync.)

Exec Function Calls

Exec function calls return string values, and are thus are one of the basic elements of string expressions. They may also appear in boolean expressions, supplying arguments for string comparisons. (A typical use of an exec function would be to return a boolean value by returning either the string 'T' or 'F'.) The form of an **exec function call** is:

```
  < <file name> [ ( <arg list> ) ]
```

where "<" is the character that signals a function invocation (just in the way that this character identifies exec files for the WorkShop's Run command). The <file name> and optional <arg list> are the same as in the SUBMIT command.

Due to our liberal conventions concerning what characters (including blanks) may appear in file names, the preprocessor must make some assumptions about how to identify the exec function file name and the argument list. Recognizing the file name is more of a problem in the case of exec functions than it for the SUBMIT command, since exec function calls may appear inside of arbitrary string expressions, while an exec invocation appears by itself in a SUBMIT command. The simple rule the preprocessor uses is: if the exec function invocation has an argument list, the file name is assumed to be everything between the "<" and the "(" beginning the argument list; otherwise, the file name is assumed to be everything between the "<" and the end of the line, which means that you will have to supply an empty argument list to an exec funtion with no arguments if the function call is not the last thing on the command line.

$IF, $ELSEIF, $ELSE, and $ENDIF

The IF, ELSEIF, ELSE and ENDIF commands allow conditional selection in exec files. The syntax of these commands is as follows:

```
  $ IF <bool expr> THEN
    <stuff>
[ $ ELSEIF <bool expr> THEN
    <stuff> ]*
[ $ ELSE
    <stuff> ]
  $ ENDIF
```

where <bool expr> is a boolean expression as described in the following section and <stuff> is made up of arbitrary normal and command lines (other than commands that would be a part of the current IF construct). The "[...]*" construct above indicates that zero or more ELSEIF commands may appear between the IF and the ENDIF command, while the "[...]" indicates that zero or one ELSE command may appear just before the ENDIF.

The IF construct is evaluated in the usual way. First, the boolean expression on the IF command itself is evaluated; if it is true then the <stuff> between the IF and the next ELSEIF (if any) or ELSE (if any) or ENDIF is selected; otherwise it is not selected. All remaining parts of the IF construct up to the ENDIF will be parsed but will not be selected once one of the <bool expr>s is true and its corresponding <stuff> is selected. To say that <stuff> is selected means that any normal lines will generate text and that any command lines will be processed. Conversely, to say that <stuff> is not selected means that any normal lines will not generate text and that command lines will be parsed (for correctness) but not executed. If the <bool expr> on the IF is not true then the following ELSEIF or ELSE will be processed. If an ELSEIF is next, its <bool expr> will be evaluated, and, if true, its following <stuff> will be selected and the remainder of the IF construct will not be selected. Processing of the IF construct continues until one of the <bool expr>s on an IF or ELSEIF is true or until the ENDIF is reached. If no <bool expr> is true before the ELSE (if any) is reached, its <stuff> will be selected.

IF constructs may be nested within each other to an arbitrary level.

Boolean Expressions -- comparison and logical operators

Booleans expressions (<bool expr>s) enable you to test of string values and check properties of files. The grammar for boolean expressions is as follows:

```
<bool expr>          ::= <bool term> [ <binary logic op> <bool expr> ]*

<binary logic op>    ::= AND
                      |  OR

<bool term>          ::= <bool factor>
                      |  ( <bool expr> )
                      |  NOT ( <bool expr> )

<bool factor>        ::= <str expr> <str op> <str expr>
                      |  <bool function>

<str op>             ::= =
                      |  <>
```

The basic element of a boolean expression (a <bool factor>) is either a boolean function (see the next section) or a string comparison, testing for equality or inequality. These basic elements may be combined using the logical operators **AND**, **OR** and **NOT**, with parentheses used for grouping. All these operators function in the usual way.

Boolean Functions -- EXISTS and NEWER

Several functions returning boolean results are provided for use with the conditional contructs.

The **EXISTS** function allows you to determine whether a file exists. The function has the following form:

```
  EXISTS ( <str expr> )
```

where <str expr> is a string expression whose value is the name of a file. Typically this <str expr> will be an expanded string constant (discussed above), such as "%1.obj".

The **NEWER** function allows you to determine whether one file is newer than another file, that is, whether its last-modified date is more recent than the last-modified date of anther file. The function has the following syntax:

```
  NEWER ( <str expr 1>, <str expr 2> )
```

where the <str expr>s specify file names. TRUE will be returned if the first file is newer than the second. A preprocessor run-time error will occur if one of the files does not exist.

$WRITE and $WRITELN

The WRITE and WRITELN commands allow exec files to write text to the console screen. This text may be used for informatory messages, prompts, or for any other purpose. The form of these commands is:

```
  $ WRITE [ <str expr> [ , <str expr> ]* ]
```

and

```
  $ WRITELN [ <str expr> [ , <str expr> ]* ]
```

That is, these commands take an arbitrary number of string expressions, separated by commas, as arguments. The strings are written to the current console line, and in the case of WRITELN a final carriage return is written.

$READLN and $READCH

The READLN and READCH commands allow exec files to read in text from the console and to assign it to a parameter variable. This mechanism may be used to obtain parameter values, to obtain values to control conditional selection, to pause until the user indicates to continue, or for any other purpose. The form of these commands is:

```
  $ READLN <%n>
```

and

```
  $ READCH <%n>
```

READLN will read a line from the console and will assign it to the specified parameter. READCH will read a single character from the console (if <return> is typed that character will be a blank).

One Restriction

Although you should not have to think about it, the preprocessor uses percents ("%") when it generates its temporary, old-style exec file. This means that you can prematurely generate and end-of-file by trying to pass two percents in a row in a normal line (both percents would, of course, have to be literalized as "~%~%").

Please let me know if you find this to be an unbearable restriction.

Examples

Example 1 -- an exec file to do a Pascal compile

This exec file does a Pascal compile and generate. Note how comments have been used to make the single-character WorkShop commands more intelligible.

```
$EXEC { "comp" — perform a Pascal compile
       %0 — the name of the unit to compile }
 P{Pascal compile}%0{source}
 {no list file}
 {default i-code file}
 <esc>{no debug file — note <esc> here represents an escape character}
 G{generate code}%0
 {default obj file}
$ENDEXEC
```

Example 2 -- an exec file to do an assembly

This exec file performs an assembly, and allows for an optional output file name which may be different from the source name.

```
$EXEC { "assemb" — perform an assembly
       %0 — the name of the unit to assemble }
       %1 — (optional) alternate name of OBJ output }
 $DEFAULT %1 TO %0 { use source name if no output name is given }
 A{assemble}%0{source}
 {no list file}
 %1{obj file}
$ENDEXEC
```

Example 3 -- a more flexible exec file to do Pascal compiles

This exec file performs compiles; it allows for an output file with a different name than the souce and permits the use of an alternate intrinsic library.

```
$EXEC { "comp1" -- perform a Pascal compile
       %0 -- the name of the unit to compile
       %1 -- (optional) alternate name for OBJ file
       %2 -- (optional) alternate intrinsic library}
 $DEFAULT %1 TO %0 { if no alternate OBJ name use same name as source }
 $IF %2 <> '' THEN { use alternate intrinsic library }
  P{Pascal compile}?{option flag}
  %2{alternate intrinsic lib}
  %0{source}
 $ELSE
  P{Pascal compile}%0{source}
 $ENDIF
 {no list file}
 {default i-code file}
 <esc>{no debug file}
 G{generate code}%0
 %1{OBJ file}
$ENDEXEC
```

Example 4 -- yet another exec file to do Pascal compiles

This compile exec file will only perform the compile if either the object file does not exist or the source file is newer than the object file (i.e., the source has changed since it was last compiled).

```
$EXEC { "comp2" -- perform a Pascal compile (only if really required)
       %0 -- the name of the unit to compile
       %1 -- (optional) alternate name for OBJ file
       %2 -- (optional) alternate intrinsic library}
 $DEFAULT %9 TO %1 { set %9 to name of output OBJ file }
 $DEFAULT %9 TO %0
 $IF EXISTS ("%9.obj") THEN
   $IF NEWER ("%0.text","%9.obj") THEN {recomp if source newer than object}
     $SUBIT comp1(%0,%1,%2)
   $ENDIF
 $ELSE { OBJ file does not exist, so generate it }
  $SUBMIT comp1(%0,%1,%2)
 $ENDIF
$ENDEXEC
```

It is left as an exercise as to how to change the above example to take into account the fact that a unit may have an arbitrary number of include files in addition to its main source file, and that the unit will have to be recompiled if one or more of these change.

Example 5 -- exec file "chaining"

This example ("make/Prog") uses the "smart" compile exec file ("comp2") defined in the last example to demonstrate how to "chain" exec file execution.

Assume we want to generate a particular program made up of three units (unit1..unit3) and that we have written "link/Prog", a smart exec file which performs a link only when one of the object files for one of the units is newer than the linked program file. Our generation exec file will use these smart exec files to perform the minimal required amount of work, thus it may be used to determine whether we have the latest version of the program without fear of wasting time.

```
$EXEC { "make/Prog" -- smart version, only recompiles & links when it has to}
  $SUBIT comp2(unit1)
  $SUBIT comp2(unit2)
  $SUBIT comp2(unit3)
  R<link/Prog              { run link exec file after compiles have run
                            so that it will get the correct file dates }
$ENDEXEC
```

Note that in the last line of the above exec file we have scheduled an exec file to be run at a later time , as opposed to SUBMITting it now, so that the file dates for the link step will be accessed after the compiles have had a chance to run. The differences between running and submitting and exec files are demonstrated in the following scenario. When an exec file is submitted it is processed immediately by the preprocessor, with its output going to the temporary file, which is then passed back to the WorkShop shell. The then shell runs the commands in the temporary file until it comes to the command to run another exec file, at which point it discards the remainder of the temporary file and runs the preprocessor with the new exec command. This exec file invocation in turn results in another temporary file of commands which is then run by the shell.

Example 6 -- a recursive exec file to do Pascal compiles

This compile exec file will perform up to 10 compiles. It takes an argument list with the names of the units to be compiled.

```
$EXEC { "rcomp" -- perform any number (up to 10) Pascal compiles.
        It calls "comp" on its first argument and then calls itself
        recursively with its arguments shifted left }
 $IF %0 <> '' THEN
  $SUBMIT comp(%0)                       { "comp" the first one }
  $SUBMIT rcomp(%1,%2,%3,%4,%5,%6,%7,%8,%9) { "rcomp" the rest, less first }
 $ENDIF
$ENDEXEC
```

Example 7 -- a Basic example

This exec file demonstrates some of the constructs in the preprocessor's meta-language, by generating the BASIC interpreter. The comments in the body of the example should be sufficient to describe what is taking place. The essential idea is that Basic is made out of three components, and that we may want to generate only one or more of them at a time.

```
$EXEC { "make/basic" -- generate the BASIC interpreter.
   There are three parameters -- if a parameter is a "Y" (yes)
   the corresponding part of the system should be generated:
     (0) the b-code interpreter
     (1) the run-time system
     (2) the command interpreter
   If no parameters are specified, the exec file will prompt to see what parts
   of the system should be generated. }
$WRITELN 'Starting generation of the BASIC system'
$IF %0 = '' AND %1 = '' AND %2 = '' THEN {no params supplied -- prompt for info}
  $WRITE 'do you want to assemble the b-code interpreter? (y or [n])'
  $READCH %0
  $WRITELN { this writeln puts us on a new line for the next prompt }
  $WRITE 'do you want to compile the run-time system? (y or [n])'
  $READCH %1
  $WRITELN
  $WRITE 'do you want to compile the command interpreter? (y or [n])'
  $READCH %2
  $WRITELN
$ENDIF
$
$IF UPPERCASE(%0) = 'Y' THEN  {assemble the b-code interpreter }
  $SUBIT assemb (int.main)
$ENDIF
$
$IF UPPERCASE(%1) = 'Y' THEN  { compile the run-time unit }
  $SUBIT comp(b.rtunit)
$ENDIF
$
$IF UPPERCASE(%2) = 'Y' OR UPPERCASE(%1) = 'Y' THEN
  ${ compile the command interpreter }
  ${ compile also if the run-time unit has changed }
  $SUBMIT comp(b.basic)
$ENDIF
$
${ link it all together }
  L{link}-p{note that "-p" gets around a linker bug}
  b.basic
  b.rtunit
  int.main
  hwintl
  iosfplib
  iospaslib


  basic{executable output}
$ENDEXEC
```

Exec Invocation Options

A number of options are available when running the preprocessor. These options may be specified when invoking the preprocessor or on SUBMIT commands. The options are specified by single letter commands following the exec parameter list. (A null parameter list should be used if you want to use options without parameters, as in "<foo()s".) The options are as follows:

"B" indicates that the preprocessor should not trim blanks on output lines. Normally the preprocessor will trim off leading and trailing blanks on the lines that it outputs to the temporary file. This allows you to indent normal lines (lines which are not exec command lines) without worrying about generating spurious blanks. Thus the preprocessor assumes that leading and trailing blanks are insignificant (which is the case for WorkShop commands, but which may not be true for some perverse programs you may run via exec files). This option will tell the preprocessor not to trim such blanks. The option applies only to the exec file being run or SUBMITted, and not to any nested exec files.

"I" indicates that the first line of the exec file is to be ignored by the preprocessor. This option is intended for deviants who like to embed their exec files in their program sources, in which case the first line of the source should be a "(*" and a "*)" should follow the end of the exec file, thus commenting it out of the program source. (Note that "(*" and "*)" should be used in preference to "{" and "}" since the latter are used as comment characters in the preprocessor.)

"T" indicates that the temporary file which is created (i.e., the expanded form of the exec file) should not be removed after it is run. One reason to use this option is to make it possible to rerun an exec file created with the step option (see below) without going through the stepping prompts a second time by running a previously created expanded exec file. The "R" exec option (described below) is used to run old temporary exec files. Note that the "T" option is not allowed on SUBMIT commands.

"R" indicates that the a exec temporary file which has been saved with the "T" option should be rerun, bypassing the normal processing by which the temporary was created. For example, "foo" may be an exec file which generates a complicated system via a large number of nested exec files which take a significant amount of time for the preprocessor to digest. If we know we are going to run "foo" repeatedly, we may want to generate the temporary file only once but run it several times. The first time we would invoke the preprocessor with "<foo()t" to indicate that the temporary file should not be automatically deleted after it is run. Subsequently, we would invoke the preprocessor with "<foo()r" to rerun the old temporary file. Note that the "R" option will override any others that may be specified, and it is not allowed on SUBMIT commands.

"S" indicates that the exec file should be processed in "Step Mode" which allows selective skipping of output lines and SUBMITs. If this option is used, the following message will appear when you invoke the preprocessor:

```
Step Mode:
-- in response to "Include ?" answer Y, N, A (Abort) or K (Keep rest).
-- in response to "Submit ?" answer Y, N, S (Step), A (Abort) or K (Keep Rest).
More details ? [No]
```

If you repond with "Y" (yes) to the "More details ?" prompt you will get further information on what each of stepping responses means.

When you invoke an exec file with the step option you will be prompted when a line has been generated and is about to go into the temporary file. The line will be displayed followed by "<= Include ?". A response of "Y" will include the line in the expanded exec file. A response of "N" will cause the displayed line to be omitted. A response of "A" will abort out of the exec file preprocessor and no exec file will be run. A response of "K" will keep (include) all the remaining lines of the exec file, leaving step mode.

When a SUBMIT command is encountered when stepping, the SUBMIT line will be displayed followed by "<= Submit ?". A response of "Y" will perform the SUBMIT unconditionally, that is, without stepping through it. A response of "N" will ignore the SUBMIT. A response of "S" will step through the SUBMIT file. A response of "A" will abort out of the exec file preprocessor and no exec file will be run. A response of "K" will keep the rest of the exec file, leaving step mode.

Note that a reponse of "?" to a "Submit ?" or "Include ?" prompt will elicit an explanation of the accepted responses.

Following are some examples of how to use the preprocessor's stepping facility.

Stepping may be used to resume execution of an exec file which did not run to termination. For example, if our example "compile" exec file includes both a compile and a generate step and if we wish to resume with the generate step we could invoke the preprocessor with "compile(foo,-work-)s". Then, in response to the "Include?" prompts for lines corresponding to the compile step we would hit "N" to skip the lines. Upon reaching the first line of the generate step we would respond with "K" to keep the rest of the file, and the generate step of the exec process would be performed.

The stepping mechanism may be used to run only selected parts of an exec file. Say, for instance, that we have a modular set of exec files which generate a whole system of programs, such as the WorkShop development system, and that one exec file called "make/all" can generate the whole system by SUBMITting exec files for each of the component programs. The exec files for each component program (development system tool) make use of other exec files to perform such standard activities as compiling (and generating) a Pascal unit or program, performing an assembly, installing a library, or manipulating files with the WorkShop's filer. If we are performing a system build and find ourselves constantly having to regenerate parts of the system due to bugs, late deliveries or whatever, then the ability to step by SUBMITs proves to be very useful. Arbitrary parts of the system can be regenerated by running "<make/all()s" (i.e., our master exec file invoked with the stepping option) and selectively submitting the sub-exec files for only those things which we wish to rebuild while stepping over the others.

Stepping in conjuction with the "T" option (for saving the temporary file created by the preprocessor) can be useful when we are going to be regenerating a single component of a program or system a number of times in succession, such as when we are fixing a bug in an element of a system build and we expect that several iterations will be needed to correct the problem. To continue our previous example, suppose that we are having a problem with the "FileIO" unit of the "ObjIOLib" library while building the development system, and that an exec file called "make/ObjIOLib" generates and installs the library, submitting compiles and assemblies for all of its units, linking everything together, and finally performing the installation. By invoking the preprocessor with "make/ObjIOLib()st" we can go into step mode and submit only those things related to the compilation of the "FileIO" unit, the link, and the installation of the library in the Intrinsic Library. Then, after each successive refinement of "FileIO", we could run the saved temporary file by running "<make/ObjIOLib()r" without having to go thru the stepping process. Our alternatives to this procedure are creating another exec file to generate only the selected parts, or running (and rerunning) the exec file for the

whole library, or running each sub-process independently (which requires more of your attention).

Exec Programming Tips

The following few points may be useful to remember when creating exec files:

Use **modular** exec files. It may helpful to think of **exec files** as **procedures** which are called via the SUBMIT command. The more modular your exec files are, the easier it will be to use the stepping facility on them.

Create **standard exec files** for common functions; for example, use one exec file to perform all your compilations. One advantage of this is that you only have to edit one file when the interface to the tool changes (as it has in the case of the assembler).

Use **optional parameters** to support features which are not always (or often) used (such as the ability to compile against an alternate intrinsic library in your compile exec file). The parameter mechanism is such that you can remain oblivious to optional parameters if you don't need the functions they support.

Write your exec files to **prompt for information** which was not supplied in parameters. This way you don't need to remember the meaning of a large number of parameters.

Exec Errors

The preprocessor can recognize a number of errors during its invocation and execution. The format in which most errors are reported is:

```
ERROR in <err loc>
<curr line>
<err marker>
<err msg>
```

where

<err loc>	is either 'invocation line' or 'line #<n> of file "<file>"'
<curr line>	is the current exec line when the error was detected
<err marker>	is a line with a question mark indicating where the preprocessor was in <curr line> when the error was detected
<err msg>	is one of the messages listed below.

IO errors are followed by an additional line with the text of the OS error raised during the IO operation. The errors detected are as follows:

IO Errors:

```
Unable to open input file "<file>".
Unable to open temporary file "<file>".
Unable to access file "<file>".
Unable to rerun file "<file>".
```

Other Errors:

```
File does not begin with "$EXEC".
End of Exec file before "$ENDEXEC".
$EXEC command other than at start.
No Exec file specified.
More than 10 parameters.
No closing ")" found.
Line buffer overflow (>255 chars).
Invalid Exec option: <option char>.
Invalid Exec option on SUBMIT: <option char>.
End of Exec file in comment.
Invalid percent: not "%n" form.
Garbage at end of command.
No argument to SUBMIT.
ELSE, ELSEIF or ENDIF not in IF.
ELSEIF after ELSE.
File contains unfinished IF.
Nothing following "<tilde>".
```

```
Out of memory. Processing aborted.
Bad temp file name generated: "<file>".
No value returned from file called as function.
RETURN with value in file not called as function.
```

and

```
Invalid command. <token> expected.
```

where <token> may be

```
String value
"%n" parameter
Terminating string delimiter
"=" or "<>"
"<>"
Boolean value
Comma (list delimiter)
"("
")"
Valid command keyword
Command
```