UniFLEX™ BASIC Precompiler User's Manual

technical systems consultants, inc.

UniFLEX™ BASIC Precompiler User's Manual

MANUAL REVISION HISTORY

Revision	Date	Change
A	10/80	Original Release, Basic Precompiler Version 1.0
B	3/81	Appendix: Add new keywords from Basic Version 2.

Table of Contents

1. INTRODUCTION

The UniFLEX™ Basic Precompiler allows programmers to produce Basic programs that are easier to read, easier to write, and result in smaller "compiled" files. (UniFLEX is a trademark of Technical Systems Consultants, Inc.) The precompiler accepts Basic source files and produces a "compiled" file similar to that produced by the "compile" command in UniFLEX Basic. This precompiler should not be confused with other Basic compilers that generate machine language code, because the UniFLEX Basic Precompiler generates an intermediate code that can only be used with UniFLEX Basic.

This is the manual for the precompiler only. It is assumed that the reader is familiar with Basic so detailed programming examples are not given nor is the syntax of the Basic language explained. These are given in the "UniFLEX Basic User's Manual".

The precompiler executes in two passes and will accept any size file on the disk for input as long as enough memory is available for internal tables. If necessary, additional memory for these tables is requested from the operating system up to the maximum permitted by the system. Two types of output can be generated. The first one is a source listing of the Basic program complete with line numbered statements and any error messages. The second is the compiled version of the program ready to be executed by Basic.

The precompiler, like Basic, performs very little syntax checking of the source statements. Most such errors will be detected by Basic when the program is executed.

This manual is organized as follows. First some terms used throughout the remainder of this manual will be defined. The next section explains how to start the program running, including a discussion of the command line parameters. Next the major features and advantages of the precompiler are explained, including precompiler control statements. And finally, the last section lists all of the error messages that can be generated by the precompiler.

2. CONVENTIONS

As in the UniFLEX Basic User's Manual, the following conventions will be used when showing the general form of a statement or command. Items not enclosed in angle brackets (<>) or square brackets ([]) are keywords and should be typed as shown. In such keywords, upper case and lower case letters are considered equivalent. Angle brackets (<>) will be used to enclose essential components of the statement. Square brackets ([]) will be used to enclose optional components.

```
<essential item>
[optional item]
```

2.1 Definitions

LETTERS

The set of letters consists of the ASCII characters "A" through "Z" and "a" through "z". The underscore character ("_") is also considered a letter.

NUMBERS

The set of numbers consists of the ASCII characters "0" through "9".

SEPARATORS

The set of separators consists of any ASCII character that is not a letter, a number, or an underscore.

PHYSICAL LINE

A physical line is defined to be one line on a terminal ending with a carriage return. It also can be thought of as one line as generated by an editor.

LOGICAL LINE

A logical line consists of one or more physical lines that contain either a single Basic statement, or several Basic statements separated by the Basic statement separators (colon or backslash). A logical line is assigned a single statement number by the precompiler and may be considered the equivalent of a numbered line in ordinary Basic. A logical line may be broken across several lines as described later on in this manual.

3. GETTING THE SYSTEM STARTED

Since there are not any built-in editing functions in the precompiler, you must have previously created a source file on disk before using the precompiler. An editor may be used to create the file. The source must be a standard UniFLEX source file, which is simply a series of textual lines terminated with carriage returns.

3.1 The UniFLEX command line

The syntax for calling the precompiler is as follows:

```
pc <source files> [+<options>]
```

where all file names are standard UniFLEX file names and default to the current working directory. Sufficient path information should be specified to accurately locate any file that is not in the current working directory.

The <source files> are previously edited files containing the Basic source lines. A single file name, or more than one file name, may be specified as containing the program to be processed. If more than one file is specified, they are processed in the order that they appear on the command line. The data contained in these files is assumed to be one complete program, perhaps broken into several parts, each part in a file. It is not possible to process more than one program with a single invocation of the precompiler.

The <options> are used to control the processing of the program and are described in detail in the next section.

3.2 Command line options

Optionally, one may include precompiler options on the command line. The list of options must start with a plus sign ("+") and may not have any embedded spaces. More than one list of options may be specified, but each list must start with a plus sign. Some of the options are single letters while others require an argument. Those that are single letters may be grouped together; for example: +bld. Those that require arguments may either stand alone or be the last of a group of options; for example: +bs10, where the "s10" is an option with an argument. For readability, those options that require an argument may use an equal sign to separate the option letter from the argument. Thus, "s10" and "s=10" are equivalent. Option letters must be specified in lower case. Following is a detailed description of each of the legal command line options.

"b" Do not create the "compiled" (binary) file.
No binary file will be created even if a binary file name is specified. This is useful when compiling a program to check for any syntax errors or for obtaining only a listing of the program.

"c" Print precompiler control statements.
Some of the precompiler control statements (described later) embedded in the source program itself are not normally printed. Specifying this option will cause them to appear in the source listing.

"d" Write "compiled" file despite errors.
If errors are detected, the "compiled" file is normally not written. Specifying this option will cause the file to be written even if errors are detected. Statements that contain errors will be incomplete and will probably cause Basic to generate an error when the program is run.

"f" Disable form feed eject.
The top of each page starts with an ASCII form feed character. Specifying this option causes no form feed to be issued. This is useful when the listing is being displayed on a terminal that uses the form feed character as a "clear screen" command.

"g" Specify load and go
If this option is specified and there were no errors detected by the precompiler, then instead of returning to the operating system when finished, the precompiler calls Basic, passing it the name of the object file. In effect, this causes the execution of the "compiled" program. This option is ignored if the "b" option (no object file) was selected. If the "d" option (produce object file despite errors) was selected, Basic will be called even if errors were detected by the precompiler.

"i" Specify line number increment.
This option requires an argument. When a program is processed, the line numbers assigned by the precompiler increment by 1 for each logical line processed. This option may be used to change that increment. For example, +i=10 may be used to set the line number increment to 10.

"l" Suppress the source listing.
If not specified, the compiler will print each line as it is processed. If this option is specified, only those lines containing errors will be printed. When this option is specified, the precompiler control

statements $lis and $nol are ignored. (These control statements are discussed later on.)

"m" Specify margin size
This option requires an argument that specifies the number of margin lines to be printed when a source listing is produced. These margin lines are blank lines printed after the page eject is performed and before the title line is printed. This option, along with the "p" option, permit the vertical centering of the printed listing on a page. The specified value must be between 0 and 255 inclusive. If this option is not specified, a default value of 3 is used.

"n" Turn off line numbers.
By default, line numbers are printed at the beginning of each logical line. But if the source program already has line numbers, like a normal Basic program, more line numbers would only be confusing. This option will cause those line numbers assigned by the precompiler to not be printed.

"o" Specify name of "compiled" file.
This option is used to specify the name of the file that is to receive the "compiled" program. The name may start immediately after the option letter "o", or may be separated from the option letter by an equal sign. For example, "+o=test" and "+otest" are both acceptable ways of specifying the file "test" as the "compiled" file. If this option is not specified, the name of the first source file followed by the characters ".bc" becomes the name of the "compiled" file. If the name of the first source file is too long to accommodate the extra characters, it is shorted to the proper length and then the extra characters are appended.

"p" Specify page length.
This option is used to specify the number of physical lines of the source program that are to be printed on a page. Each page consists, therefore, of this number of source lines, plus 5 lines for the title and subtitle areas, plus any margin lines. The specified value must be between 1 and 255 inclusive. If this option is not specified, a default value of 55 is used.

"s" Specify starting line number.
Normally, the precompiler starts assigning numbers to the logical lines starting with 1. This option may be used to specify that a different starting line number be used. For example, specifying "+s=10" or "+s10"

indicates that the first line number is to be 10.

"t" Suppress title and margin printing.
This option causes the precompiler to not attempt to format the listing of the source. No title lines are printed, no margin lines are printed, and the number of lines per page value is ignored. In addition, all precompiler control statements are printed and the control statements $pag and $spc are ignored. No page ejects are performed. This option finds its greatest use when the source listing is routed to a disk file for later viewing in case it is needed for troubleshooting.

Some examples of these options are:

```
++ pc progl +lo=test  no listing
                      "compiled" file is "test"

++ pc p1 p2 +nb       listing on
                      no line numbers
                      no "compiled" file

++ pc test +cds10 +i10
                      list source command statements
                      write "compiled" file despite errors
                      starting line number is 10
                      line number increment is 10
                      "compiled" file will be "test.bc"
```

3.3 Printer interface

The precompiler does not have a built-in method to output to a hardcopy device. However, since the program listing is routed through the UniFLEX standard output device, the "pipe" mechanism may be used to route the output to a printer spooler. For more information, consult the UniFLEX Operating System Manual.

4. FEATURES

Several things stand out as the main features of the Basic Precompiler. They are:

(1) unlimited length variable names,
(2) unlimited length label names,
(3) continuing logical lines across physical line boundaries,
(4) embedded comments in the source program, and
(5) compilation and listing format control via precompiler control statements.

4.1 Variable Names

Variable names may be of any length and may contain letters, numbers, and the underscore character ("_"). The first character must be a letter or an underscore, and the name must be followed by a blank, separator, or the end of the logical line. Upper and lower case letters are distinct. Thus, the variable "first_time" is considered different from "First_time". The name cannot be the same as one of the Basic keywords (a list of them is in an appendix to this manual). All upper and lower case variants of keywords are also forbidden. Thus, "open", "Open", "OPEN", etc. are all illegal variable names. However, string and integer variants of keywords are legal; eg. "open%" and "open$" are legal variable names even though "open" is not. If the keyword is itself a string or an integer (eg. chr$), then the floating point and integer (or string) variants may be used as variable names. For example, "chr$" is a keyword, but "chr" and "chr%" are valid variable names.

Also, the name of a floating point variable cannot start with the letters "fn" (or any upper or lower case variants) unless it is a call to or definition of a user-defined function. Integer and string variables, however, may begin with "fn". In the case of user-defined functions, upper and lower case variations of the letters "fn" refer to different functions. Thus, "fna" and "Fna" are distinct user-defined functions. Here are some examples of variable and function names:

```
THIS_IS_A_VARIABLE_NAME
So_is_this
this_is_a_STRING_variable$
_SO_IS_THIS_$
this_is_too$
FNCTION_IS_A_FUNCTION_NAME
_THIS_IS_AN_INTEGER_VARIABLE%
_SO_IS_THIS_%
response$
```

```
        mode%
```

Some illegal variable names are:

```
        1_cannot_start_a_variable_name
        9CANNOT_START_A_VARIABLE_NAME_EITHER
        CLOSE          (variable names cannot be keywords)
```

4.2 Line Labels

Basic normally requires an integer line number on every source line of the program. The precompiler on the other hand, only requires a label on a line to which the program will transfer control. Also, the label need not be an integer, it can be any contiguous series of characters consisting of letters, numbers and underscores. Any other character terminates the label name. All statement labels must begin in column one, and statements must start in column two or beyond. Some examples are:

```
THIS_IS_A_LABEL REM This is a remark with a label

1000            REM That was the label "1000"

this_is_a_label_without_a_statement

0000    REM Note that 0000 is a legal label name, therefore
        REM "goto 0000" is a valid statement; but
        REM "goto0000" references the variable "goto0000"
```

Labels may appear in expressions. They are translated into integer constants by the precompiler. Thus if the label "restart_line" has been assigned the value 120 by the precompiler, then the statement:

```
            if erl<>restart_line then on error goto 0
```

is a valid statement. In this case, the system variable "erl" would be compared to 120.

4.3 Continuation of Lines

The precompiler allows logical lines to be split across physical line boundaries or in other words, a logical line may consist of one or more physical lines. To do so, just place a backslash ("\") before the carriage return. The precompiler converts the backslash-carriage return combination to a space. This means that variable names and keywords cannot be continued onto the next line since the space is a separator character. It should be noted that multiple spaces and horizontal tab characters are ignored except inside of strings where they are significant. For example:

```
      IF DELTA% <= GAMMA% THEN PRINT 'DELTA ='; DELTA% \
                               ELSE PRINT 'GAMMA ='; GAMMA%

   * DEFINE RECORD I/O BUFFER

      FOR I=0 TO NUMBER_ELEMENTS :            \
          FIELD #1, I*ELEMENT_SIZE AS G$,     \
                    15 AS FIRST_NAME$(I),     \
                    15 AS LAST_NAME$(I),      \
                    09 AS SOC_SEC_NU$(I),     \
                    02 AS INDEX$(I)   :       \
      NEXT I
```

In the first line of the example, the "if-then-else" statement is considered as one logical line even though it is split across two physical lines. It should be pointed out that a remark statement after the "then" portion would cause the "else" statement to be ignored, since remarks stop at the end of the logical line. The next logical line is the line that begins with an asterisk in column one. (The blank line is ignored.) This is a comment line and is ignored by the precompiler. The last logical line consists of seven physical lines starting with the "for" statement and ending with the "next" statement. Even though seven physical lines are involved, only three Basic statements are used (the "for", "field", and "next" statements). If more than one statement is on a logical line, the statements must be separated by either a colon (:) or a backslash (\). A backslash-carriage return combination does not act as a statement terminator.

4.4 Embedded comments in the source program

It is possible to embed comments in the source program at any point outside of quoted strings. The comment is enclosed in braces ("{" and "}"). Comments may span physical lines and do not need the "backslash followed by carriage return" convention to be continued. For example:

```
   dim a(3,4), {This could describe the use of this matrix}\
       b(4,5), {This comment could describe the second matrix
                and could be continued on another line without
                using the backslash }\
       c(3,5)  {Note that backslashes are needed
                outside of comments}
```

4.5 Precompiler control statements

As the above example shows, any line that starts in column one with a separator is considered to be a comment line. In most cases, a comment line is ignored by the precompiler. If the comment starts with a dollar sign ("$") in column one then the line is considered to be a precompiler control statement. Spaces may appear between the dollar sign and the control statement. There are five groups of precompiler control statements:

(1) string/macro definition statements
(2) conditional compilation statements
(3) variable type declarations
(4) pagination and listing control statements
(5) miscellaneous statements

4.5.1 String/macro definition statements

The precompiler allows the user to assign a string to an identifier. All subsequent references to the identifer in the source program will be replaced by the specified string. Arguments may be passed when the string is used, and the definition may be changed or removed.

4.5.1.1 Creating and using definitions.

Definitions are created using the "$def" statement. The general form of the $def statement is:

```
$def <name>=<string>
```

The name is any combination of letters and/or digits as defined earlier. The string is any series of characters. It is not necessary to enclose the string in quotation marks. The string may span several lines, but the line continuation convention (backslash followed by carriage return) must be used. Thus, a definition may not include more than one logical line. If the specified name has already been used for a definition, the previous definition is removed and the new definition used. Here are some examples of definitions:

```
$def random_number=8.*rnd(0)+1
$def clear_array=for i%=0 to 10:\
                 a(i%)=0:\
                 next i%
```

In the first example, the name "random_number" was defined to be the string "8.*rnd(0)+1". Any occurrence of the name "random_number" in the program subsequent to the definition would cause the name to be replaced by "8.*rnd(0)+1". An example of its use is:

```
if random_number <= 4 then x=x+1
```

The second example shows a multi-line definition. Note, however, that it spans only physical lines, not logical lines. As with normal lines in the precompiler, the backslash followed by carriage return is replaced by a single space when the line is processed. Since this definition can be interpreted as a complete logical line, it may stand alone in the program. For example:

```
clear_array
```

It may also be used as part of a larger construction, for example:

```
if b<>0 then clear_array
```

It is also possible to pass parameters when calling a definition. Within the definition itself, a substitutable parameter is indicated by an ampersand (&) followed immediately by a digit from 0 through 9 inclusive. The first substitutable parameter is indicated by &0; the last, by &9. When calling the definition, the actual values are passed by enclosing each one in a pair of square brackets ([]) immediately following the name of the definition. As an example, we will modify the "clear_array" definition to accept parameters.

```
$def clear_array=for i%=&1 to &2:\
                 &0(i%)=0:\
                 next i%
```

This definition allows us to specify the name of the array to be cleared, and the index bounds of the portion of the array to be cleared. Here is an example of its use:

```
clear_array[a][3][7]
```

This would cause the following statements to be generated:

```
for i%=3 to 7:\
a(i%)=0:\
next i%
```

A null argument may be specified by merely specifying the square brackets, for example: abc[]. If insufficient parameters are specified when the definition is called, no substitution takes place for those references that do not have a corresponding parameter.

There are some limitations that must be observed when defining and using these defined strings.

A definition cannot include anything that must start in column 1. Thus, a definition cannot contain labels or precompiler control statements. A call to a definition cannot start in column 1; this column is reserved for labels.

The name of a definition cannot be the same as a Basic keyword.

A search for a definition is made before searching for a variable name or line label. Thus, definition names may be the same as variable names and line labels, but this would disable the use of the variable name or line label until the definition is removed. This holds true even if the variable name is that of an integer or string. Thus, if there is a definition "abc", then the variables "abc", "abc%", and "abc$" could not be used.

Definitions may contain calls to other definitions, but a definition should not call itself.

Lastly, calls to definitions may not appear within Basic strings (enclosed in single or double quotation marks).

4.5.1.2 Removing definitions

A definition is removed by using the $undef control statement. The general form of this statement is:

```
$undef <name>[,<name>...]
```

The names are those of the definitions that are to be removed. No error is generated if a name is specified for which a definition does not exist.

4.5.2 Conditional compilation statements

Names defined with the $def control statement may be used to effect the conditional compilation of parts of the source program. Precompiler control statements which test for a name having been defined are used to determine if a segment of the program is to be included or excluded. The following is a discussion of each of the conditional compilation control statements.

$ifdef <name>
$ifndef <name>
: The $ifdef statement asks if "name" is currently defined (as the subject of a $def statement). The $ifndef statement asks if "name" is currently not defined. If the statement is true ("name" is defined for $ifdef or not defined for $ifndef), then all lines following the statement up to a $orifdef, $orifndef, $else, or $endif statement are processed. If the test is false, then those lines are skipped.

$orifdef <name>
$orifndef <name>
: These statements provide for alternative tests. If a preceding test ($ifdef, $ifndef, $orifdef, or $orifndef) was false, then these statements are evaluated. If the statement is true ("name" is defined for $orifdef or not defined for $orifndef), then all

lines following the statement up to a $orifdef, $orifndef, $else, or $endif statement are processed. If the test is false, then those lines are skipped. As soon as any one of a consecutive series of $ifdef, $ifndef, $orifdef, and $orifndef statements is true, subsequent ones will be skipped until a $endif statement is found.

$else
The $else statement is used to indicate the final alternative to a sequence of $ifdef, $ifndef, $orifdef, and $orifndef statements. If none of the previous tests were true, then all lines between the $else statement and the $endif statement are processed. If any previous test was true, all lines between the $else and the $endif are skipped.

$endif
The $endif is used to indicate the end of a conditional.

Here is an example of the use of conditionals:

```
*    The following statement defines "aaa"
*    to be the null string.
$def aaa=

$ifdef bbb
 rem This will be skipped since "bbb" is not defined
$orifdef aaa
 rem This will be processed since "aaa" is defined,
 rem even if it is only the null string.
$orifndef bbb
 rem This will be skipped because even
 rem though "bbb" is not defined,
 rem a previous test was true.
$else
 rem This also will be skipped because a previous
 rem test was true.
$endif
```

4.5.3 Variable type declarations.

Normally, a variable is considered to represent a floating point number unless it is followed by a special character to indicate an integer variable or a string variable. If most of the variables used in a program are integers or strings, typing the percent signs or dollar signs can become quite burdensome. The precompiler allows the programmer to specify which of the three variable types, floating point, integer, or string, is to be assumed if there is no special character appended to the name. This is achieved through the $type control statement. The general form of this statement is:

```
$type <type letter>
```

The type letter is either an "i", an "s", or an "f", indicating integer, string, or floating point, respectively. The type letter may be in either upper case or lower case. When a $type statement is encountered, all variables that do not have a special character appended to them are assumed to be of the specified type. When the default type is set to either string or integer, floating point variables may be specified by appending an exclamation point to the variable name. For example:

```
$type i   {Define default type to be integer}
 a=1      {This refers to the integer variable a%}
 a$="pdq" {The string variable requires the dollar sign}
 a!=pi    {This refers to floating point variable "a"}

$type s   {Now change the default type to string}
 a="abc"  {Now "a" refers to a string variable.}
 a%=2     {The integer variable "a" now requires the
           percent sign}
 a!=5.    {The floating point variable still requires the
           exclamation point}

$type f   {Change to floating point}
 a=10.    {Floating point variables now do not need the
           exclamation point}
 a%=10    {Integers need the percent sign}
 a$="10"  {Strings need the dollar sign}
```

Notice that more than one $type statement may appear in a program. The specified type remains in effect until another $type statement is encountered. There is one limitation that must be observed when using the $type statement. If a variable is the same as a Basic keyword except for a trailing special character (eg. open$), then it must always be specified with the trailing special character. The special character is the only way that the precompiler knows that the name is a variable and not a keyword.

The precompiler also allows the programmer to specifically declare that certain variables have a specific type. When they are so declared, these variables do not need a special character appended to their names to indicate the type, regardless of the value of the default type. The types are declared using one of the three declaration statements $float, $string, and $integer. The general forms of these declaration statements are:

```
$float <name>[,<name>...]
$integer <name>[,<name>...]
$string <name>[,<name>...]
```

As an example, let us assume that a program uses a lot of floating point variables, and only uses integer variables for loop control and subscripts. The programmer decides to reserve the variables i%, j%, and k% for these variables. To avoid having to type the percent signs every time that a loop variable or subscript is used, the following statement may be used:

```
$integer i,j,k
```

Any time that the variables i, j, and k are used without any special characters appended to them, they are assumed to refer to the integer variables. Of course, these variables may also be specified with trailing percent signs since they are really integers. The $float and $string declaration statements are used in a similar manner. Names that are the same as Basic keywords should not be specified in type declarations since the precompiler needs the trailing special character (percent sign or dollar sign) as an indication that the name is a variable and not a keyword.

Arrays may also be declared to have a specific type. In this case, the characters "()" are appended to the variable name. For example,

```
$string x,y(),z
```

In this example, the variables x, and z, and the array y are declared to be of type "string". No dimension information should be specified in the type declaration; that is done through the Basic "dim" statement.

One limitation to the declaration of types of specific variables is that the declarations must occur at the front of the program, before any Basic statements are processed. The declarations may be preceded only by other precompiler control statements, blank lines, and comments (those that start with a separator in column 1, not Basic remarks).

4.5.4 Pagination and listing control statements

Pagination and listing control statements allow the programmer to format the printed listing of the program. The following are descriptions of those control statements.

$lis
: The $lis control statement is used to resume the listing of the source program after it had been turned off by the $nol control statement. If listing is already taking place, then the $lis control statement is ignored. This control statement is also ignored if the "l" command line option was specified. This control statement is not normally printed in the source listing unless the "c" or "t" command line option was specified.

$nol
: The $nol control statement is used to turn off the listing of the source program. If listing has already been turned off then the $nol control statement is ignored. This control statement is not normally printed in the source listing unless the "c" or "t" command line

option was specified.

$pag
The $pag control statement causes a page eject to occur. Normally, a page eject is automatically performed whenever a page is filled, but by using the $pag command one can cause a page eject to occur earlier. This control statement is ignored if the "t" command line option is specified, and is not normally printed in the source listing unless the "c" or "t" command line option was specified. If the source listing is already at the top of a page, this command is ignored.

$spc <n> [, <m>]
The $spc control statement causes <n> blank lines to be inserted into the listing. Optionally the <m> parameter can be specified which is a keep count. If there are less than <m> lines left on the page then instead of spacing <n> lines, a page eject is performed and processing of the space command is terminated. This is useful to prevent a block of lines from being split across a page. This control statement is ignored if the "t" command line option is specified, and is not normally printed in the source listing unless the "c" or "t" command line option was specified. This command is also ignored if the source listing is at the top of a page.

$sttl [string]
The $sttl control statement sets the program sub-title to the specified string. The sub-title may be 0 to 80 characters long. If the string is longer than 80 characters, anything past the 80th character is ignored. The sub-title string is printed left justified under the title line. If no sub-title string is specified, the sub-title is set to spaces. This control statement is not normally printed in the source listing unless the "c" or "t" command line option was specified.

$ttl [string]
The $ttl control statement sets the program title to the specified string. The title may be 0 to 35 characters long. If the string is longer, anything past the 35th character is ignored. The title string is printed left justified on the same line as the date and page number. If no title string is specified, the title is set to spaces. This control statement is not normally printed in the source listing unless the "c" or "t" command line option was specified.

4.5.5 Miscellaneous control statements.

$lib <file name>
The $lib control statement tells the precompiler to start reading the source from another disk file. The <file name> should be in the normal UniFLEX format, with sufficient path information to accurately locate the file. These alternate input files may be nested up to 12 deep. (This may result in more open files that the operating system

allows.)

If path information is specified, only that path is searched for the file. If no path information is specified in the name of the file, a series of directories is searched in an attempt to find the file. First the current working directory is searched. If the file is not found there, then a subdirectory named "lib" in the current working directory is searched. If that subdirectory does not exist, or does not contain the file, then the directory "/lib" is searched. If the file cannot be found in any of these directories, an error message is issued.

When all of the statements in the alternate file have been read, input reverts back to the file that contained the $lib statement.

$scale <n>

The $scale control statement sets the Basic scale factor to <n> where "n" is between 0 and 6 inclusive. For example:

```
$scale 3
```

This control statement must precede any Basic statements in the source file. It may be preceded only by other precompiler control statements, blank lines, and comments (those that start with a separator character in column 1, not Basic remarks). If an error occurs in the $scale statement, the scale factor is set to zero. See the UniFLEX Basic User's Manual for information on the proper use of the scale factor.

5. Error Messages

There are two types of error messages that can be generated by the precompiler. The first type is from errors found on the command line calling the precompiler from UniFLEX. These errors also include those detected when making the first pass over the source program. These errors are fatal, causing the precompiler to terminate immediately. They are:

Erroneous page length/margin value
: A page length of zero or a page length or margin value larger than 255 was specified, or a non-digit was encountered in the specified value.

Error in line number start/increment
: A non-digit was encountered in the argument to the "s" or "i" option.

Line number overflow
: A line number was generated by the precompiler that is larger than 32767. If a line number increment or starting value was specified on the command line, their values should be made smaller so that the largest line number generated by the precompiler does not exceed 32767.

Line number start/increment is zero or too large.
: The argument to the "s" or "i" option was zero or greater than 32767.

Memory overflow. Translation terminated.
: Not enough memory was available for the precompiler to make an entry in one of its internal tables. More space may be made available by reducing the number or length of variable names, label names, and definitions.

Missing object file name
: The "o" command line option was seen but was not followed by a file name.

No files specified
: No input file names were specified on the command line.

Program larger than 65535 bytes.
: The program is too large to fit into memory.

Unknown option specified
: An unknown option character was found after the plus sign.

Errors detected when trying to open the source files are reported along with the name of the file that generated the error.

The second type of errors are source code errors.

Array declaration not followed by "()"
A variable in a $string, $float, or $integer control statement was followed by a "(" but not by "()".

Bad constant
An error was detected when trying to convert an ASCII number to binary. The number could be too large, too small, or contain an illegal character.

Cannot open library file
A file referenced in a $lib control statement could not be found or opened.

Declaration not before first statement
A $string, $float, or $integer control statement was encountered that was not before the first Basic statement. Declarations may be preceded only by other precompiler control statements, blank lines, and comments.

Dummy variable may not be typed
A dummy variable was specified to a user-defined function that had a trailing dollar sign, percent sign, or exclamation point.

Duplicate declaration
The same variable name was given two types by appearing in conflicting $string, $float, or $integer statements.

Duplicate line label
Two lines have the same label.

"Endif", "orif", or "else" without corresponding "if".
A $endif, $else, $orifdef, or $orifndef was encountered when there was no previous $ifdef or $ifndef.

"If" nesting level too deep
The maximum nesting level for conditional statements is 255.

Illegal "defined" name
An illegal name was specified as the argument to a $ifdef, $ifndef, $orifdef, or $orifndef control statement.

Illegal separator in name
A separator character was detected in a variable name in a

$integer, $float, or $string control statement.

Illegal scale factor
The scale factor was too large, too small, or the scale command was not on the first line of the original source file.

Illegal type specified
A $type control statement was encountered that did not specify either "i", "f", or "s".

Missing equal sign in definition
An equal sign could not be found after the name in a $def control statement.

Missing quotation mark
A closing quotation mark was missing from a string constant.

No "]" found
No closing bracket was found after an argument to a definition call.

String too long
A string constant is limited to 255 characters.

Unbalanced Parentheses
An expression has unbalanced parentheses.

Undefined line label
A reference was made to a label that does not exist.

Unrecognizable character
A character was seen that has no meaning to the precompiler. This may be caused by having too many percent signs or dollar signs at the end of a variable name. In general, if the precompiler is expecting to find a variable name or label name and finds a separator character instead, it will issue this error message. Control characters other than carriage return and horizontal tab appearing outside of quoted strings also cause this error.

Appendix

The following is a list of all keywords defined in UniFLEX Basic. These may not be used as variable names.

ABS	GOSUB	RESPONSE
AND	GOTO	RESTORE
ARG$	HEX	RESUME
ARGC%	IF	RETURN
AS	INCH$	RIGHT$
ASC	INPUT	RND
ATN	INSTR	RSET
CHAIN	INT	SECOND
CHD	KILL	SEEK
CHR$	LEFT$	SGN
CLOCK$	LEN	SIN
CLOSE	LET	SIZE
COMMON	LINE	SLEEP
COS	LOG	SPC
CVT$%	LSET	SQR
CVT$F	MEM	STEP
CVT%$	MID$	STOP
CVTF$	MODE	STR$
DATA	NEW	STRING$
DATE$	NEXT	SWAP
DEF	NOT	TAB
DIGITS	OLD	TAN
DIM	ON	TASK$
ELSE	OPEN	TERM$
END	OR	THEN
ERL	PI	TIME$
ERR	POS	TO
ERROR	POSITION	TSTAT%
EXEC	PRINT	UNLOCK
EXIT	PUT	USING
EXP	RANDOMIZE	VAL
FIELD	READ	WAIT
FN	RECORD	WIDTH
FOR	REM	
GET	RENAME	