# TIOLIB
# User's Guide

**Timothy D. Pointon**

**Sandia National Laboratories**

Last updated: January 4, 2010

# Table of Contents

## 1.0 Overview

TIOLIB is a FORTRAN subroutine library that handles I/O functions typically required of a program. It is designed primarily to provide a powerful, free-format, command-line interface for large programs. These commands have the form:

```
command_name parameter1 ... parametern
```

where the parameters can be character strings, integers or real numbers in any order, separated by any number of delimiter characters, and distributed over one or more input lines. TIOLIB was originally designed primarily for interactive applications, but can be easily applied to a non-interactive program reading its input from a text file.

This document describes how to use the TIOLIB input line processor, and is applicable to all applications using TIOLIB for input. In addition to processing the free-format command lines, TIOLIB provides control features for processing of sets of application commands:

1. Definition of scalar and/or array symbols. The expression operator evaluates either a simple symbol substitution or an arithmetic expression involving multiple symbols and/or constants.
2. Running command files with passed arguments.
3. Running loops.
4. Conditional command processing.
5. Command archiving.

These functions (and others) are controlled by a set of ***I/O control commands***, separate from the main application's commands, processed internally by TIOLIB. TIOLIB uniquely differentiates between application and I/O control commands by requiring that I/O commands be prefixed with the ***I/O control command prefix character.*** By default this character is "^". Thus, for example, to use TIOLIB's ***case*** command to convert all input characters to lower case, you would enter,

```
^case lower
```

No space is required between the prefix character and the command name. As described below, TIOLIB provides for continuation lines. However, it is strongly recommended to always put the I/O command on the same line as the prefix character, because some of I/O commands require this restriction. Since I/O commands **must** be prefixed with the control character, there is never ambiguity between application and I/O commands.

TIOLIB further generalizes this convention of using prefix control characters for commands by supporting ***auxiliary command sets***. An auxiliary command set is a set of commands used by the application but processed elsewhere. Each auxiliary command set has its own command prefix character, initially defined by the application. When TIOLIB encounters one of these

characters, it transfers control to the routines that handle the auxiliary command set rather than the main application. For example, suppose an application linking to the TIO library also uses a graphics library that has its own set of commands for controlling its features, and provides routines to process those commands using the TIO library. By designating '#' for the graphics command prefix character and providing a simple interface routine, any such application linked with TIOLIB and this graphics package automatically has three sets of commands:

| | |
|---|---|
| `command_name [arg_list]` | is a main application command |
| `^command_name [arg_list]` | is an I/O command |
| `#command_name [arg_list]` | is a graphics option command |

The command prefix characters can be changed interactively by the user with the **char** I/O command.

## 1.1 Command Line Syntax

The basic unit of input returned by TIOLIB to be processed by the application is a ***command line***. A command line consists of one or more lines of input, denoted as 'sublines', connected with ***continuation characters***. Each subline contains ***tokens***—alphanumeric strings, separated from each other by an arbitrary number of ***delimiter characters***. All characters following the ***comment character*** in a subline are treated as a comment, and ignored. Blank lines and comment lines (i.e., lines in which the first non-delimiter character is the comment character) are ignored, and can be embedded within a command line. By default, the following characters are used:

| | |
|---|---|
| **continuation** character | & |
| **delimiter** characters | SPACE and TAB |
| **comment** character | ! |

The application can also select additional characters to be delimiters (typically "," and "="), and the user can change the continuation and comment characters with the ***char*** command (use the ***show char*** command to see what characters are in use.) The following example illustrates a command line:

```
command_name param1 param2 &
param3 & ! Embedded comment #1
! Embedded comment #2
param4
```

TIOLIB parses an input command in two stages. The first stage is to concatenate the "active" part of each subline, i.e., the text up to a comment or continuation character, or an end-of-line, into a single FORTRAN character variable. In the example above, this processing would result in:

```
command_name param1 param2 param3 param4
```

Note that white space between two parameters across a continuation line is significant. In this example, if there had been no white space between the last non-blank character of ***param$_2$*** and the continuation character, and before ***param$_3$*** on the next line, the command to be processed would have only three parameters:

```
command_name param1 param23 param4
```

where ***param$_{23}$*** is a concatenation of the two text strings ***param$_2$*** and ***param$_3$***.

The second stage of processing an input command takes the FORTRAN character variable from the first stage, and construct the "expanded" input line actually used to supply input tokens to the application. The characters in the *expanded* input line are built using the rules:

1. All sequences of delimiter characters are replaced with a single space.
2. All 'input tokens' are converted to their final 'output token' form.

The input tokens in a command line can either be a simple sequence of non-delimiter characters, a quoted string, a TIOLIB expression (either a symbol, command file parameter, a loop variable, or an arithmetic combination of these), or a concatenation of the above types. To build final output tokens, the following special characters are used:

| | |
|---|---|
| **literal** character | \ |
| **quote** characters | " and ' |
| **expression** character | $ |

The following rules, based loosely on UNIX shell processing, govern the building of output tokens:

1. The ***literal*** character causes the next character to be put into the token (but not the \ itself), ignoring any special function it may have.
2. Characters outside quoted strings and expressions are converted to the current TIOLIB case (see the ***case*** command).
3. TIOLIB expressions (either simple symbols, command file parameters, and loop variables; or an arithmetic combination of these) are evaluated and replaced with their equivalence value, using the syntax **$expression**, **${expression}**, or **$(expression)**. Character strings returned from the expression parser are concatenated with any leading or trailing non-delimiter characters in the raw input line.
4. Case conversion is disabled within either single or double quoted strings. Note that the quotes of a quoted string are part of the token itself (the routine processing the token is responsible for deciding whether or not to actually remove the quotes).
5. Single quoted strings disable all special character functions, except for \ (thus \' can be used to put a single quote in a single quoted string).

6. Within double quoted strings, expression substition is performed. Double quoted strings can also be used to quote a string with a single quote in it. Note that when processing is complete, double quoted strings are converted internally to single quoted strings, and any single quote character in the string itself is converted to \'.

Note that single or double quoted strings must be on a single line, because the routine that initially reads in the command requires that quotes are closed on each *subline*. However, longer strings can be constructed by concatenation operations.

The following examples illustrate the building of a token from an input string. In these examples, it is assumed that TIOLIB is currently converting to lower case, and that the symbols ***sym1*** and ***sym2*** have the equivalence strings of ***hello*** and '***abc 123***' respectively (the quotes for *sym2* are not part its symbol value).

| **Input Token** | **Output Token** |
|---|---|
| `123.5` | `123.5` |
| `UP_down` | `up_down` |
| `'Quoted String'` | `'Quoted String'` |
| `$sym1` | `hello` |
| `\$sym1` | `$sym1` |
| `"$sym1 world"` | `'hello world'` |
| `'$sym1 world'` | `'$sym1 world'` |
| `s${sym1}w` | `shellow` |
| `"${sym2}'56"` | `'abc 123\'56'` |
| `"$sym2"` | `'abc 123'` |
| `$sym2` | `abc 123` |

The last two examples illustrate an important feature of expression evaluation. Inside double quotes, evaluation results in a single quoted string token. Outside quotes, ***sym2*** expands into two output tokens. Since all input tokens are processed before the line is returned to the application, a symbol whose value has delimiters in it will result in more than one output token. In fact, it is possible to define a symbol to represent an entire command (see Section 3.1 below).

## 2.0 Expression Evaluation

TIOLIB provides several ways of defining either scalar or array variables, described in detail in the following sections. When building the *expanded* input line, if TIOLIB encounters the expression evaluation character "$", it transfers control to the expression evaluation routine. There are three forms of a TIOLIB expression:

```
1. $expression
2. ${expression}
3. $(expression)
```

The expression parser loads characters from the raw input line until either it:

1. encounters a delimiter character for the first form.
2. finds a matching "}" or ")" for the leading "{" and "(" for forms 2 or 3 respectively. With these two forms, any number of delimiter characters can be used within the outer pair of braces or parentheses.

We define a "simple" expression to be a single variable name. Prior to the May 2003 release, this was the only option in TIOLIB, and only the first two forms were used. The second form was only necessary when concatenating a variable's text string with trailing non-delimiter characters, as illustrated in Section 1.1.

With the May 2003 release of TIOLIB, the user can define "complex" expressions. The most important use of this feature is to substitute numeric values for arithmetic expressions of scalar TIOLIB variables and constants. The expression parser uses the following syntax rules:

1. All expressions are defined using standard *infix* notation, with sub-expressions delimited by pairs of parentheses.
2. Within an expression, any unquoted string is interpreted as a TIOLIB variable, and any numeric value as a constant. Note that after the expression character, you **do not** precede subsequent TIOLIB variable names with a "$".
3. Array elements use subscripts of the form "`[i]`", starting at i = 0 for the first element. If the array subscript is a sub-expression, it must evaluate to an integer.
4. Literal strings can be used in expressions with single quotes around them.

For example if *a, b* are scalar TIOLIB variables, *c* is a TIOLIB array of length 4, and *i* is a scalar TIOLIB variable with integer values between 0 and 2, the following are legal expressions

```
1. $a+b, ${a+b}, ${a + b}, or $( a + b )
2. $((a+1)*(c[2]-2)), or ${ (a + 1) * (c[2] - 2) }
3. $( (a+b+c[i+1]) / ((a+2.0)*c[2]) )
```

In the first example, we do not need the delimiting {} or () if there are no spaces between any characters in the expression: "`$a+ b`" is illegal because the expression parser would terminate loading characters after encountering the space between the "+" and the "b". The second and third examples are expressions that require either the second or the third form; they need to be bounded by an extra set of () or {}. Using the first form on the second example, `$(a+1)*(c[2]-2),` the expression parser would terminate loading characters after the ")" following the "`+1`".

In general, an expression is built from combinations of unary and binary operator constructs using the syntax:

1. *unary_op*(operand)
2. (operand$_1$ *binary_op* operand$_2$)

Operands can either be simple scalar variables, or array elements, `array_name[index]`. For array elements, the index can itself be an expression, as long as it evaluates to an integer within the array bounds (0 to *n*-1).

The recognized **unary** operators are:

| | |
|---|---|
| int | - Convert to INTEGER (truncate) |
| nint | - Convert to nearest INTEGER (round) |
| rea | - Convert to REAL |
| chs | - Change sign |
| sqrt | - Square root |
| exp | - Exponential |
| ln | - Natural logarithm |

The recognized **binary** operators are:

| | |
|---|---|
| `+ - / *` | - Basic arithmetic operators |
| ^ | - Raise to power |
| mod | - Modulus |
| max | - Maximum of two arithmetic values |
| min | - Minimum of two arithmetic values |
| token | - Extract the n'th substring from a string with spaces in it: string token n. |

TIOLIB variables do not have explicit types—they are all character strings. However, when evaluating an expression, the type of each operand is determined as it is processed, according to the following hierarchy:

1. Integer
2. Real: floating point numbers with no exponent, or an ‘e’ or ‘E’ exponent.
3. Double precision: floating point numbers with a ‘d’ or ‘D’ exponent.
4. Alphanumeric string

Unary operators return results of the same type, except that *sqrt, exp,* and *ln* return a *real* when operating on an i*nteger*. When binary operators process operands of mixed type, the result is returned as the higher type of the two operands. The type conversion operators allow the user to explicitly define the type of the result. It is only necessary to use these going from a higher type to a lower type (*e.g.*, an integer is a legal real number). REAL results are saved with six decimal place precision, while DOUBLE PRECISION results use thirteen. Note that the only legal operations on character strings are "+", which results in the concatenation of the two operands, and the token operator.

To illustrate expression substitution, we will assume the following variable definitions (see Section 3 for more details):

```
^define i 3
^define a 3.0
^define b 4.0
^define c [ 1 2 3 4 ‘abc’ ]
```

The following examples illustrate expressions formed with these variables:

| Expression | Result String | Result Type |
|---|---|---|
| $i+1 | ‘4’ | INTEGER |
| $nint(a/b) | ‘1’ | INTEGER |
| $a+b^i | ‘6.70000E+01’ | REAL |
| $(2*(a-1) + b/3.) | ‘5.33333E+00’ | REAL |
| $(2*(c[2]-c[0]) + c[3]/3) | ‘5’ | INTEGER |
| $(c[i+1] + ‘DEF’) | ‘abcDEF’ | ALPHA |

In the fourth example, all arithmetic is done in integer, and so the result c[3]/3 is truncated to 1. The last example illustrates string concatenation. The expression parser is actually a clumsy way of concatenating strings. Since TIOLIB merges the result of an expression with any trailing non-delimiter characters in the raw input line into a single token in the expanded input line, a more efficient way of achieving the same result is:

`$(c[i+1])'DEF'` *or* `${c[i+1]}'DEF'`

Finally, it should be noted that some applications may use a comment character that is a binary expression operator. For example, suppose an application sets the default comment character to be '*'. In this case, each instance of the '*' character intended for use as a binary operator must be prefixed by the *literal* character.

## 3.0 Symbols

TIOLIB symbols are defined with the ***define*** or ***gdefine*** command, and deleted with the ***undefine*** command. The ***define*** command defines a "local" symbol, available only to the current command file level (see Section 4). When a command file defining a symbol with the ***define*** command finishes, the symbol is automatically deleted. The ***gdefine*** command defines a "global" symbol; one that can only be deleted with the ***undefine*** command. Global symbols defined in a command file are available to all command file levels; either at higher levels if the file runs another command file, or the lower level when the file finishes.

## 3.1 Scalar Symbols

Scalar symbols are defined using the following form of the *define* or *gdefine* command:

```
[g]define symnam arg1 [additional arguments and operators]
```

The first parameter after *symnam* is required. If it is the only parameter, it becomes the symbol value. If any additional parameters are present, the remaining parameters after *symnam* are treated as input to an RPN calculator. The "calculation" must evaluate to a single operand on the stack. Historically, this was the only way to do arithmetic operations with symbols, and is preserved to support old input files. With the new version of TIOLIB, symbol values can be also defined using infix expressions. The following two commands are equivalent

```
^define ktsnp $dtsnp $dt / nint $ktav 1 - 2 / +
^define kthis $(nint(dtsnp/dt) + (ktav-1)/2)
```

In fact, the result of these two equivalent forms are *identical*, because the infix expression is first converted to the RPN form, and then uses the same calculation routine for evaluation.

Using quoted strings, symbols can be defined with embedded spaces. If the symbol value in a *[g]define* command is a quoted string, TIOLIB removes the quotes before storing the value associated with the symbol. When these symbols are evaluated, they result in multiple tokens in the command line returned to the application. For example, suppose that an application has a command named ***window*** that requires four real parameters. The following sequences of commands all produce exactly the same result

```
1. window 1.2 2.5 -3.2 7.9

2. ^define w1 '1.2 2.5 -3.2 7.9'
   window $w1

3. ^define w2a '1.2 2.5'
   ^define w2b '-3.2 7.9'
   ^define w2 "$w2a $w2b"
   window $w2

4. ^define w 'window 1.2 2.5 -3.2 7.9'
   $w
```

Finally, note that it is possible to define a symbol value to be a quoted string, by simply adding an extra set of quotes in the definition:

```
^define quostrng "'A quoted string'"
```

TIOLIB only removes the outer set of quotes. A second way to construct a command with a quoted string parameter is to simply put double quotes around the symbol. Thus the commands,

```
^define title 'Plot Title'
title "$title"
```

are equivalent to the following command,

```
title 'Plot Title'
```

## 3.2 Symbol Arrays

There are two ways to define a symbol array:

```
[g]define arrnam  [ elem1 elem2 ... elemn ]

[g]define arrnam  [ n ]
```

The brackets here are required syntax to specify an array, not an indicator of optional input. The first form defines an array of *n* elements, with an explicit value for each element. Each element can be a TIOLIB expression. For example, the following command defines a local array of four elements,

```
^define a [ 1 2.0 $(c+d) 'a string' ]
```

As with scalar-valued symbols, each array element is just a character string that does not have an explicit type unless it is used in an expression. Thus array elements do not have to be of homogeneous type. The second form of the ***[g]define*** command simply defines an array without specifying the value of any elements. Individual array elements can then be defined as if they were scalar symbols, using the command:

```
[g]define array[index_expression] arg1 [args_and_operators]
```

This command has one syntax restriction: `array[index_expression]` must parse as a single input token—there cannot be any spaces between the array name and the left bracket, or between the index expression and the brackets. There is no such restriction in the array definition command, because the brackets are parsed as separate input tokens. For example, to construct a global array of integers 1 - n, the following commands can be used:

```
^gdefine a [ $n ] ! Space between brackets OK here
```

```
^for i 1 $n
  ^gdefine a[$(i-1)] $i ! No spaces in "a[$(i-1)]"
^endfor
```

Note that the command defining the elements matches the command defining the array; if the array is global, you must *gdefine* the elements.

There are two special operations involving array-valued symbols. The first is a special unary function returning the number of elements of an array:

```
$n_elements(a)
```

This function cannot be used in arithmetic expressions, which only handle scalar operands. However, for the *special case* that there are exactly two expression tokens, with the first one being the unary function `n_elements`, and the second a TIOLIB variable, it returns the number of elements in that variable. If the operand is a scalar variable, the return value is 1. Although this special expression can be used anywhere an ordinary expression can, it is typically used as the equivalence string of a scalar variable:

```
^define nelem_a $n_elements(a)
```

This function is useful in command files, where it is used to obtain the length of array-valued parameters passed by reference (see Chapter 4.0 below).

The second special operation is the simple expression `$a`, where `a` is an array name. In this case, the expression value is a space-delimited list of all elements in an array. For example,

```
^define a [ 1 2 3 4 5 ]

command $a
```

is equivalent to

```
command 1 2 3 4 5
```

In this simple example, we could also have accomplished the following with

```
^define a '1 2 3 4 5'

command $a
```

However, the length of a symbol value is limited to 132 characters, while the array method can be used with an arbitrary number of elements.

## 4.0 Command files

TIOLIB allows applications to run command files. For a non-interactive application using TIOLIB for input, running a command file effectively *includes* the contents of the file in the input stream. Command files can contain any commands recognized by the application, *i.e.*, application, I/O control commands, or auxiliary commands. A command file is invoked with the I/O ***run*** command. The command format is:

```
^run command_file_name [parameter1 ... parametern]
```

Command file parameters are optional. When TIOLIB parses the *run* command, it removes outer quotes from parameter strings before storing them, just as in symbol value processing. Thus command file parameters can have multiple-token values. Their values can be used in a command file in one of two ways:

1. Using a convention based on UNIX shell processing, in which the i'th parameter is a read-only value, substituted using a simple expression of the form `$i` or `${i}`. This was the only way to access parameters prior to the May 2003 release, and remains the default.

2. With the May 2003 release of TIOLIB, symbolic names can now be assigned directly to the parameters, and they can be either passed by either value or reference. To invoke this new passing mechanism, the *first executable line* of a command file must be the new I/O command:

   ```
   ^cmfparam name1 ... namen
   ```

   The symbolic names are assigned to the parameters passed by the calling command. To pass parameters by reference, use TIOLIB's "address of" character "@" ("&" conflicts with the default *continuation* character), as in the following example:

   ```
   ^run command_file param1 @param2 @param3 param4 @param5
   ```

   As indicated, pass-by-value and pass-by-reference parameters can be mixed.

The new version of TIOLIB continues to support the original format, for processing legacy files. However, there are limitations with the old form. First, the parameters are restricted to *read-only* scalars; passing an array as a command file parameter can only be done by reference. Second, you cannot directly use an old-style command file parameter in a complex expression, because the expression parser treats integers as numeric constants (only for the case of a simple expression does it treat an integer as a command file parameter "name"). To use an old-style parameter in a complex expression, you must first define a temporary local variable with the parameter's value.

Command files can be nested up to four deep. The primary input source—terminal input for interactive applications, or the main input file for non-interactive applications—is defined as

“level 0”. Command files invoked from here are *level 1*, command files invoked from these are *level 2*, *etc*. Local symbols are available only to the command file that defines them. Within a command file, parameters passed to it are handled as local symbols. Before executing the first command in the file, local *read-only* symbols are created with the passed values, associated with default “names” of “1”, “2”, *etc*. If the first executable command is “`^cmfparam ...`”, the default names are replaced with the user-defined ones. If a parameter is passed by reference, the local symbol becomes an alias for the symbol passed to the file, and its read/write status is switched to *read-write*. These parameters are redefined with the usual “`^define name value`” command. A pass-by-reference parameter does not need to exist when a command file is called, but the file is required to define a value in this case.

The following simple example illustrates an old-style command file:

```
!  File: LOAD_AND_PLOT.CMD
!
$1
plot $2 $3 $4
```

Invoking this file with the command:

```
^run load_and_plot.cmd ‘load  a’ a 20 "’Title’"
```

results in the execution of the two commands

```
load a
plot a 20 ’Title’
```

The next example illustrates the new format: a command file that creates an array of n elements, loaded with n!. The first parameter is read-only, and can be passed by either value or reference, while the second can only be passed by reference.

```
!  File: BLD_FACT_ARR.CMD
!
^cmfparam n a

^define a [ $n ]

^define a[0] 1
^for i 1 $n-1
  ^define a[$i] $((i+1)*a[i-1])
^endfor
```

Here is an example of calling this file:

```
^run  bld_fact_arr.cmd 10 @fact10
```

If the local symbol `fact10` did not exist, this command would create it. If it did already exist, it's previous value would be overwritten. In the latter case, there is no restriction on whether the previous value was a scalar or an array. The old value is discarded and replaced with the new one.

Finally, we describe error-handling in command files. By default, if an error occurs in a command file, the file is closed, and control returns the next lower command file level. The `cmferr` TIO command provides user control over this behavior. The error can either be ignored, the current command file closed (default), or all command file levels closed and control returned to zero level input. Note that if an error occurs processing the new `cmfparam` command (*e.g.,* a mismatch in the number of parameters), the command file is unconditionally closed.

## 5.0 Loops

TIOLIB supports simple *for* loops for repetitive processing of commands. TIOLIB *for* loops use the following syntax,

```
^for loopvarnam min max [step]
  {loop body}
^endfor
```

All loop limit parameters must be integer (the `nint` function can be used to generate integer arguments from a calculation involving real-valued operands), and ***step*** defaults to 1 if omitted. The ***for*** and ***endfor*** commands must be on the same line as their I/O command prefix character. Input lines within the loop body access the loop variable's current value using the variable name in an expression. Here is an example of a simple loop:

```
^def n 10

^for i 1 $n
  command a $i $(2*i+1)
^endfor
```

Loops can be nested up to four levels deep at each command file level. Loop variable names are local to the command file level running the loop. To use these variables in a command file called within the loop, the user must pass then down either as a command file parameter, or by defining global symbol with the current loop variable value.

Although loops are usually defined in command files, they can be run directly from interactive input. After entering the "`^for ...`" command, TIO returns with a "`FOR>`" prompt. All lines entered will be put into the loop body, until an "`^endfor`" command is encountered. The loop will then execute.

## 6.0 Conditional Statements

TIOLIB supports two types of *block-if* conditional constructs. The first is based on whether or not a TIO symbol is defined, while the second type of construct is for algebraic comparisons of two operands.

The first type of conditional statements use the `ifdef` and `ifndef` commands (just like the C preprocessor, cpp), as shown in the examples below:

```
^ifdef a [then]
  {commands}
^endif
```

and

```
^ifndef a [then]
  {commands}
^endif
```

The trailing `then` keyword is optional (see the *logical* `if` syntax below). In the first case, the commands will be executed only if symbol ***a*** is defined, while in the second case, the commands are executed only if symbol ***a*** is not defined. Note that you ***do not*** put a $ in front of the symbol name.

The second type of conditional construct is the *logical* `if` command, which uses a logical comparison between two operands, and executes the commands if the test is true. The syntax of the basic `if` command is illustrated with the following example:

```
^if $i eq 1 then
  {commands}
^endif
```

The commands will be executed only if the value of symbol ***i*** is equal to 1. Note the following syntax requirements on the TIO *logical* `if` command line. First, the logical comparison must not be enclosed in parentheses. Second, the terminating `then` keyword on the end of the line is required (to distinguish between basic and compound logical expressions, described below). There are six logical comparison operators: ***eq, ne, lt, gt, le, ge***. Two tests can be concatenated with the logical operators "***and***" and "***or***":

```
^if $i ge 1 and $i le 10 then
  {commands}
^endif
```

Note that more complicated *inline* logical tests are not possible because TIOLIB does not allow parentheses in the command line. If necessary, more complicated tests can be accomplished by creating temporary symbols. Tests are done in the highest type of the two operands. Mixed integer/float comparisons are done in real arithmetic. Floating-point tests should be done with some care to avoid roundoff errors. Only the ***eq*** and ***ne*** operators are valid for strings.

The TIOLIB `else` command enables multiple *block-if* constructs, selecting a single set of commands to be executed from two or more. The general form of the construct is:

```
^ifdef if_condition1
  {command set 1}
^else if_condition2
  {command set 2}
...
^else if_conditionn
  {command set n}
^else
  {command set n+1}
^endif
```

where *if_condition*$_i$ is either an `ifdef, ifndef` or *logical* `if` test, and the last unconditional set of commands is optional. If an *if_condition* is true, the corresponding set of commands will be executed. Once the command block has executed, control "drops through" to the first command line after the `^endif`. The following example illustrates a multiple *block-if* construct:

```
^if $i eq 1 then
  {commands}
^else if $i ge 2 and $i le $j then
  {commands}
^else ifdef k then
  {commands}
^else
  {commands}
^endif
```

Note that all I/O control commands for conditional processing require that the command be on the same line as the I/O prefix character. Also note that there must be a space between `else` and `if`, because they must parse as separate tokens (`elseif` is not a recognized keyword).

Finally, we point out a current limitation for setting up input files with conditional constructs. All conditional control commands are internally *expanded* by the command line parser, **even** if they are nested inside a block of commands that does not get executed. Thus all TIOLIB variables in every *logical* `if` command must actually exist, or the command line parser will return a fatal error before it is ever processed. Thus, if you want to set up a single file with two distinct

options, each using several variables that are not defined with the other option, be aware that you should not use *logical* `if` commands with the variables specific to one option. In future releases, we will remove this limitation, not expanding the TIOLIB control lines for a conditional construct nested inside an outer conditional block that does not get executed.

## 7.0 Command archiving

TIOLIB provides a command archiving feature—saving all user commands to an external file, which may or may not be supported by the actual application. To see if archiving is supported, use the ***show archive*** command; archiving is supported if the application provides an archive file name. At any point, the user can toggle archiving with the ***archive*** command. Note that only commands that successfully execute will be archived. The decision about whether the command executed "successfully" is made by the code that processes it. Thus TIOLIB makes the decision for I/O commands, but the application or auxiliary command set processor are responsible for their own commands, not TIOLIB. When done running the application, the archive file can be renamed (it is a hard-wired file name!), and used as a command file to recreate the session (note that some minor editing of the archive file may be required).

## 8.0 I/O Command Dictionary

The following list briefly describes the available I/O commands:

| | |
|---|---|
| **archive** | Control command archiving |
| **case** | Control case conversion of incoming characters |
| **char** | Set values for TIOLIB control characters |
| **cmferr** | Command file error handling |
| **cmfparam** | Define command file parameters used in a command file |
| **define** | Define local symbol |
| **echo** | Control echoing of expanded input lines |
| **for** | Execute **for** loop |
| **gdefine** | Define global symbol |
| **if** | Conditional test on comparison of two operands |
| **ifdef** | Conditional test on TIOLIB symbol being defined |
| **ifndef** | Conditional test on TIOLIB symbol not being defined |
| **pause** | Interrupt command file or for loop execution |
| **prompt** | Set interactive prompt |
| **run** | Run command file |
| **show** | Display TIOLIB internal status |
| **spawn** | Spawn a sub-process |
| **undefine** | Delete a symbol definition |
| **wide** | Toggle terminal width (80 or 132 characters). |

For each command, the user need only enter enough characters to uniquely identify the command. Thus **sp** is sufficient to invoke the **spawn** command. Note however, that **^endfor** must be entered exactly to terminate a **for** loop. A detailed description of each command is given below.

## ARCHIVE

Controls archiving of user input. The following command formats are possible:

| | |
|---|---|
| **ARCHIVE off** | Turn off archiving |
| **ARCHIVE on [cmflvl]** | Archive commands up to command file level ***cmflvl*** (Default is ***cmflvl*** = 0) |
| **ARCHIVE all** | Archive all commands |

Note that the application must support archiving by supplying TIOLIB with an archive file name. To verify that this is so, use the ***show archive*** command.

Default: No archiving

## CASE

Controls the case of non-quoted characters returned by TIOLIB.

format: **CASE case_switch**

Where ***case_switch*** takes the values *lower, upper,* or *none.*

Default: lower case

## CHAR

Changes special characters used in TIOLIB input processing.

format: **CHAR char_type character**

The following character types are recognized:

| | |
|---|---|
| **comment** | - Comment character |
| **continuation** | - Line continuation character |
| **io_command** | - I/O command character |
| **aux(i)** | - i'th auxiliary command character (note that **aux** by itself is interpreted as **aux(1)** ) |

The following characters are reserved by TIOLIB, and cannot be used

**\ ' " $ { } SPACE TAB** together with any other application-defined delimiters

Defaults:

| | |
|---|---|
| **comment** | **!** |
| **continuation** | **&** |
| **io_command** | ^ |
| **aux(i)** | ***Application specific*** |

## CMFERR

Sets the handling of errors encountered in command files (see Chapter 4.0 above).

format: **CMFERR err_option**

Where **err_option** takes the values

| | |
|---|---|
| **continue** | - Ignore error and continue running file |
| **close_cur** | - Close current command file |
| **close_all** | - Close all command files in current chain |

Default: **close_cur**

Note that for certain severe errors, the ***continue*** option may be overriden, e.g., errors linking passed and declared command file parameters with the ***cmfparam*** command, or

errors loading the command body of a ***for*** loop.

## CMFPARAM

Define symbolic names for the parameters passed to a command file.

format: **CMFPARAM $param_1$ $param_2$ ... $param_n$**

This command must be the first *executable* command of a command file (preceding comment lines are not counted).

## DEFINE

Define a scalar or array-valued symbol whose value(s) is available only to the current command file level. For a scalar symbol, the format of this command is,

**DEFINE symnam {value}**

If **{value}** is a single token, it is the symbol value. If it consists of more than one token, it is used as input to an RPN stack calculator (see Section 3.1) to compute the symbol value.

For an array-valued symbol, there are two command formats:

**DEFINE arrnam [ $elem_1$ $elem_2$ ... $elem_n$ ]**
**DEFINE arrnam [ length ]**

The brackets here are required syntax to indicate an array (*i.e.*, they do not denote optional input). The first format explicitly defines the array elements. The second simply defines the length of array; each element can be defined using a scalar ***define*** command.

Local symbols are automatically deleted when a command file finishes, or they can be explicitly deleted with the ***undefine*** command.

## ECHO

Toggles echoing of 'expanded' input lines returned by TIOLIB.

format: **ECHO on/off [level [ioecho [cmtecho [line_width]]]]**

The 'expanded' line is the full input line (including any continuation lines) with all **for** loop variables, command file parameters, and TIOLIB symbols replaced with their equivalent strings. The optional switches after ***on*** are:

**level** - Lowest level of command file that will be echoed. Default is 1, i.e., all command file lines are echoed, but not zero level (interactive) input lines.

**ioecho** - Control switch for echoing I/O command lines (0 = NO, 1 = YES). Default = 1

**cmtecho** - Control switch for echoing comment lines (0 = NO, 1 = YES). Default = 0

**line_width** - # characters in the echo output lines (default is # lines on terminal, or 80 if echo unit is a file)

Default: No echoing

## FOR

Execute a **for** loop. See Chapter 5.0 above for a description of **for** loops.

## GDEFINE

Define a scalar or array-valued *global* symbol, *i.e.,* one whose value(s) is available to all command file levels.

**GDEFINE symnam {value}**

If **{value}** is a single token, it is the symbol value. If it consists of more than one token, it is used as input to an RPN stack calculator (described in detail in Section 3.1 above) to compute the symbol value.

For an array-valued symbol, there are two command formats:

**GDEFINE arrnam [ $elem_1$ $elem_2$ ... $elem_n$ ]**
**GDEFINE arrnam [ length ]**

The brackets here are required syntax to indicate an array (*i.e.,* they do not denote optional input). The first format explicitly defines the array elements. The second simply defines the length of array; each element can be defined using the scalar g***define*** command.

Unlike local symbols, global symbols are not deleted when a command file finishes — they can only be deleted with the ***undefine*** command.

## HELP

Provide documentation for TIOLIB and all its commands. If the environment variable "TIOhelp" is defined and specifies an existing file, it will be used as the help file. If this is not the case, TIOLIB then looks for the file "$HERMES_ROOT/doc/Tiolib.pdf", which will be used as the help file. The help file is opened by spawning a new process with the command "acroread", unless the environment variable "TIO_help_reader" is used to define an alternate command for reading the help file. If the help file is not located by either of these methods, a list of available TIOLIB commands will be provided.

## IF

Conditional control statement based on logical comparison tests. See Chapter 6.0 above for a description of TIOLIB conditional statement processing**.**

## IFDEF

Conditional control statement based on whether a TIOLIB symbol is defined. See Chapter 6.0 above for a description of TIOLIB conditional statement processing.

## IFNDEF

Conditional control statement based on whether a TIOLIB symbol is *not* defined. See Chapter 6.0 above for a description of TIOLIB conditional statement processing.

## PAUSE

Interrupt command file or FOR loop execution.

format: **PAUSE [switch]**

The **pause** command is only useful if default input is from the terminal. It is used in two ways:

1. When input is directly from the terminal, the **pause** command is used to set a flag controlling whether **pause** commands in loops or command files will be enabled. **switch** takes the values *on* or *off*.

2. Inside a command file or loop, a **pause** command by itself, (i.e., no **switch** argument) will interrupt execution and prompt the user to continue, ***but only*** if the pause switch is turned on. Note that the user can enter the command 'quit' in response to the pause prompt to kill the currently running command file or loop.

By default, the pause switch is enabled. Note that the switch ***cannot*** be toggled except when input is from command file level zero (i.e., the terminal, since the pause command is basically useful only for interactive input).

## PROMPT

Set the prompt for interactive input.

format: **PROMPT newprompt**

Default: *****

## RUN

Run a command file.

format: **RUN command_file_name [parameter1 ... parameter_n]**

See Chapter 4.0 above for a more detailed description of command files.

## SHOW

Display TIOLIB information.

format: **SHOW item**

Where **item** can take the values

| | |
|---|---|
| **archive** | - Display archive information |
| **char** | - Display main characters, and auxiliary command sets |
| **symbol** | - Display symbol information |

## SPAWN

Spawn a sub-process

format: **SPAWN [sub_process_prompt]**

The prompt is an optional parameter. If not supplied, a default prompt set by the application will be used.

## UNDEFINE

Delete one or more symbols.

format: **UNDEFINE $sym_1$ [$sym_2$ ... $sym_n$]**

If $sym_1$ = '*', all *local* symbols at the current command file level will be deleted, leaving global symbols untouched. *Global* symbols can only be deleted explicitly by name.

Note that in earlier versions of TIOLIB, the undefine command supported another format, "`undefine n`", which would delete the last `n` symbols that had been defined. This format is no longer supported, and will generate an error.

## WIDE

Toggle terminal width.

format: **WIDE [on/off]**

The terminal width is either 80 or 132 characters  If no switch is supplied, the terminal width is toggled from its current state to the other.  *on* selects 132 character width, and *off* selects 80.  Note that this command only works for terminals that support (and are enabled to accept) the ANSI escape sequences for changing the terminal width.

Default:  80 character terminal width

## 9.0 TIOPP – A General-Purpose Macro Preprocessor

The TIO library provides a utility for processing files containing TIO symbols, loops and conditional constructs. This provides a fast and efficient method for taking advantage of TIOLIB's capabilities using existing software without having to modify that software's input processing to use TIOLIB. Instead, TIOPP can be used to translate an input file utilizing any of TIOLIB's capabilities into a file suitable for use by the unmodified existing software. Even when a software tool uses TIOLIB to process its input, TIOPP can be used to find and repair mistakes in TIO syntax and usage in input files without the complication of simultaneously dealing with errors associated with the application's command usage.

Specifically, TIOPP performs the following operations while processing its input:

1. Symbol definition and substitution, including arithmetic manipulation of both integer and floating point symbols.
2. Conditional command processing, allowing selected blocks of commands to be enabled or disabled in the output file.
3. Expansion of loops, allowing sets of related commands to be written out in an easy-to-read format.
4. Expansion of command files with passed arguments. This allows users to modularize their input using multiple files.

## 9.1 Command-line options

TIOPP is invoked from the command line, and its behavior and initial state can be modified using command-line options and arguments. The syntax for invoking TIOPP is as follows:

**tiopp** [ **-D** *tio_var*[**=***value*] ... ] [**-f**] [**-c** *cmt_char* ] [ **-w** *columns* ] [ **-e** *max_err* ] \
[ *input_file* [ *output_file* ]

| | |
|---|---|
| **-D** *tio_var*[**=***value*] | Predefine TIO global symbol *tio_var* with *value*. If *value* is not supplied, the value of *tio_var* is set to 1 (the numeral "one"). |
| **-f** | Overwrite existing output file. By default, TIOPP issues a warning, and interactively prompts whether or not to overwrite the file. |
| **-c** *cmt_char* | The character to be interpreted as the comment character by TIO. If not supplied, '!' is used. |
| **-w** *columns* | The maximum number of characters that will be written to a single output line. The default is 76 characters. |
| **-e** *max_err* | Maximum number of errors allowed before TIOPP quits. Default value is 20. |
| *input_file* | Name of input file to be processed. If not supplied, input is read from standard input. |
| *output_file* | Name of output file. If not supplied, output is written to standard output. |

TIOPP can be safely run non-interactively if the **-f** option, input file, and output files are all specified on the command line. It returns a zero exit status if no errors were detected, or one if any errors were detected processing the input file.