PowerParser Documentation

Release 2.0

Chuck Wingate and Bob Robey

December 20, 2015

CONTENTS

Contents:

CHAPTER

ONE

ABOUT

PowerParser capabilities go far beyond simple parsing and setting of variables. For *PowerParser*, the parsing concept is extended to think of the user input as a set of commands to control the operation of the code.

PowerParser handles command line arguments and interprets the user input file. It does the following:

- Parses the exectuable command line
- Reads the user input file.
- Compiles the user input file, handling variables, math expressions, and loops
- Provides functions for extracting data from the compiled file

Some of the extended features of *PowerParser* include:

- Count before reading
- Multi-line comments
- User defined variables
- Math Expressions
- Intrinsic functions
- If statements
- Loops
- Subroutines
- Include other files
- Execution line arguments
- Checking input

PowerParser is a general purpose input file parser with some simple programming capabilities such as mathematical expression evaluation and loops

Authors: Bob Robey XCP-2 brobey@lanl.gov Chuck Wingate caw@lanl.gov

works, such modified software should be clearly marked, so as not to confuse it with the version available from LANL.

Additionally, redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the Los Alamos National Security, LLC, Los Alamos National Laboratory, LANL, the U.S. Government, nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE LOS ALAMOS NATIONAL SECURITY, LLC AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL LOS ALAMOS NATIONAL SECURITY, LLC OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

This is LANL Copyright Disclosure C15076/LA-CC-15-054

CHAPTER

TWO

GETTING STARTED

Download the *PowerParser* package, currently PowerParser_v2.0.7.tgz. The tests use the genmalloc package, genmalloc_v2.0.7.tgz, to allocate contiguous 2D arrays for testing the *PowerParser* code. So download that if you want to run the tests.

Untar the genmalloc and PowerParser files:

```
tar -xzvf genmalloc_v2.0.7.tgz
tar -xzvf PowerParser_v2.0.7.tgz
```

The build processs uses cmake and is similar for both packages. If building with genmalloc, modify the CFLAGS and LDFLAGS variable to include the path to the genmalloc build products:

```
export CFLAGS="${CFLAGS} -I<genmalloc_path>/include
export LDFLAGS="${LDFLAGS} -L<genmalloc_path> -lgenmalloc
```

Run the cmake command as follows:

```
cmake .                          // in-tree build
cmake <path-to-src>              // out-of-tree build
cmake -DCMAKE_BUILD_TYPE=debug <path-to-src>
cmake -DCMAKE_BUILD_TYPE=release <path-to-src>
cmake -DCMAKE_INSTALL_PREFIX=/usr/local // set the install directory
```

Then build the package:

```
make
```

And install:

```
make install
```

CHAPTER

THREE

USER'S GUIDE

This User's Guide describes the general parsing capabilities of the *PowerParser* package from an application user's perspective. Since this is just a library, the full capabilities of using *PowerParser* are dependent on the integration of the parsing package into a particular implementation. Application developers using *PowerParser* should supplement this with the particular behavior of their package with the user input.

We start with the basic syntax recognized by *PowerParser* and move on to more advanced features that require tighter integration into the application.

3.1 Scalar Variables

The basic input in the user input file is:

```
some_command = input values
```

PowerParser will search through the user input file for every occurence of "some_command" and will set internal variables to the input values. An example of setting a simple scalar value is:

```
dx = 1.0
```

This will set the internal variable in the code corresponding to "dx" to 1.0. Often the internal variable name will be the same as the command name, but it is not required to be so.

The types of variables that can be set are:

- real
- integer
- character
- boolean
- logical

Examples of these are:

```
xreal   = 1.0
ivalue  = 100
string  = "malarky"
doThing = true
```

The logical input is either true or false. Various forms of these are allowed. For example:

```
true, .true., True, FaLse, false, ...
```

3.2 Arrays

These examples are for C/C++ style indexing with a base index of 0. This style is set by including set_index_base_zero somewhere in the input file. The default style uses a base index 1, which is more familiar to Fortran style programmers.

Arrays are input by adding the index (or indices) to the command name. For example, to input a one-dimensional (1D) array, use:

```
array1d(0) = 1.0, 2.3, -5.6, 7.1e19, 3, -3.4e-23
```

The index (the number in parentheses) is the starting point in the array to fill with values. In the above example, position 0 in array1d would be set to 1.0, position 1 would be 2.3, position 2 would be -5.6, etc.

If the index is not supplied, then it is assumed to be 0 (or 0,0 for 2D, 0,0,0 for 3D, etc). The following example without the index works the same as the previous example:

```
array1d = 1.0, 2.3, -5.6, 7.1e19, 3, -3.4e-23
```

The index can be any positive integer, as long as the values you input do not exceed the array length. For example the above example could be rewritten as:

```
array1d(0) = 1.0
array1d(1) = 2.3, -5.6
array1d(3) = 7.1e19, 3, -3.4e-23
```

The commas separating the values are optional. Space delimited values are also allowed. For example:

```
array1d(0) = 1.0  2.3  -5.6  7.1e19  3  -3.4e-23
```

The comma character can also be used as the line continuation character. See the Multi-line Commands Section for a description of the continuation characters.

Multi-dimensional arrays are input the same as 1D arrays, but with more indices. And example of 2D input is:

```
array2d(0,0) = 1.0  2.3  -5.6  7.1e19  3  -3.4e-23
```

The first array index varies the fastest (as in Fortran), then the second index, etc. The user has to know the bounds on all the array indices except the last one.

In the above example, for instance, if the first dimension bound is 3, then the command is equivalent to:

```
array2d(0,0) = 1.0
array2d(1,0) = 2.3
array2d(2,0) = -5.6
array2d(0,1) = 7.1e19
array2d(1,1) = 3
array2d(2,1) = -3.4e-23
```

But if the first dimension bound were 2, then it would be equivalent to:

```
array2d(0,0) = 1.0
array2d(1,0) = 2.3
array2d(0,1) = -5.6
array2d(1,1) = 7.1e19
array2d(0,2) = 3
array2d(1,2) = -3.4e-23
```

An example of a 4D integer array follows, the dimensions of the first 3 indices are 2, 1, and 3, the comments are an aid in keeping track of the indices.

```
! 0 0 0 0
! 1 0 0 0
! 0 0 1 0
! 1 0 1 0
! 0 0 2 0
! 1 0 2 0
! 0 0 0 1
! 1 0 0 1
! 0 0 1 1
! 1 0 1 1
! 0 0 2 1
! 1 0 2 1
i4d(0,0,0,0) = 3.0 4 -14 49
i4d(0,0,2,0) = 19 42
i4d(0,0,0,1) = -3 542 -165 555
i4d(0,0,2,1) = 199 942
```

Following is an example of a 3d logical array where the first two dimensions are 3 and 2.

```
! 0 0 0
! 1 1 0
! 2 0 0
! 0 1 0
! 1 0 0
! 2 1 0
! 0 0 1
! 1 1 1
! 2 0 1
! 0 1 1
! 1 0 1
! 2 1 1
! 0 0 2
! 1 1 2
! 2 0 2
! 0 1 2
! 1 0 2
! 2 1 2
log3d(0,0,0) = true false false false true
log3d(2,1,0) = false true false false
log3d(0,1,1) = false true false true &
true true false false true
```

Arrays of character strings are also allowed. For example:

```
char1d(0) = "May", "the" "force"
char1d(5) = "you"
char1d(3) = "be" "with"
```

Note that the above array specification is is out of order in that position 5 is specified before positions 3 and 4. This is allowed and is equivalent to:

```
char1d = "May" "the" "force" "be" "with" "you"
```

The quotes are not required and the above example can also be written as:

```
char1d(1) = May the force be with you
```

Of course if you want spaces or special characters in your strings you must use the quotes. For example, setting the title as one string is:

```
title = "May the force be with you"
```

If the maximum number of characters allowed in the string is exceeded by the user input, then the user input is truncated to match the maximum number of characters.

The single quotes character, ‘, can be used instead of the double quotes. The two types of quotes function identically. The title could also be written as:

```
title = 'May the force be with you'
```

However, mixing the types of quotes for any single string is not allowed, the following produces a fatal error.

```
title = 'May the force be with you"
```

3.3 Parsing Syntax and Comments

Understanding the parsing behavior of *PowerParser* is important in understanding how the input file is interpreted. This includes multiple commands on a line, multi-line commands, duplicate commands and comments.

3.3.1 Multiple Commands on a Line

Semicolons can be used to separate commands on a single line. The four line example in the Scalar Variables section could also be written as:

```
xreal   = 1.0; ivalue  = 100; string  = "malarky"; dothing = true
```

3.3.2 Multi-line Commands

When inputing an array of numbers the usual continuation character is the ”,” character. For example the following two lines:

```
i_array(0) = 1, 3, 4,
             5, -9, 13*10
```

will be essentially merged together and treated as a single line, i.e.

```
i_array(0) = 1, 3, 4, 5, -9, 13*10
```

If in the original two lines, the ”,” following the 4 was removed, as in

```
i_array(0) = 1, 3, 4
             5, -9, 13*10
```

then, *PowerParser* will not merge the two lines and the second line would generate a fatal error.

PowerParser also allows another continuation character, the “&” character, for example:

```
$hyp = (sqrt($dx * $dx + &
             $dy**2))
```

These two lines are combined and processed as one line,

```
$hyp = (sqrt($dx * $dx + $dy**2))
```

See the Math Operations Section for a detailed discussion of math operations.

The "&" continuation character can also be used in array specifications, as in:

```
i_array(0) = 1, 3, 4 &
             5, -9, 13*10
```

The "," characters are not really necessary (space delimited numbers work as well as as, delimited numbers) and this could also be written as:

```
i_array(0) = 1 3 4 &
             5 -9 13*10
```

3.3.3 Multiplicity

When several values are repeated, for example in an array command, the multiplicity operator, "*", can be used to save space and typing. For example, the following input:

```
i_array(0) = 1 3 5*4
```

expands to

```
i_array(0) = 1 3 4 4 4 4 4
```

The multiplicity operator, "*", should not be confused with the multiplication operator, which is also "*". The multiplication operator will only be found where math expressions are allowed. For example:

```
$var1 = (5*4)
```

which sets the variable "$var1" to 20 (see the Variables Section for a description of variables). The basic rule for math expressions is that anything in parentheses is evaluated as a math expression. See Math Operations for a detailed discussion of math expressions.

The multiplicity operator is why math can only be done in math expressions, such as inside parentheses, (...). Consider the command:

```
i_array(0) = 3*4
```

Without the rule about math only being done in math expressions, such as inside parentheses, (...), the above command is ambiguous and could be interpreted as "4 4 4" or as "12".

3.3.4 Duplicate Commands

Commands can occur multiple times in the input file. For example:

```
doThing = true
... other commands
doThing = false
... other commands
doThing = True
```

PowerParser will extract the last occurrence. So in the above example, "doThing" will end up being true. Warning messages will be issued when the same command is found multiple times in the input file. This can be turned into a fatal error.

There are two general types of comments allowed: single line comments and multi-line comments. The single line comments treat everything from the single line comment character to the end of the line as a comment. The multi-line

comment has beginning and end comment characters with everything between those two comment characters being treated as a comment.

Three single line comment characters are available in *PowerParser*. They are "!" (Fortran style), style# (Shell style), and "//" (C++ style). The "//" comment is composed of two characters but it is treated as a single comment delimiter. Following are a few examples of single line comments:

```
some_cmd = 5.0 ! Some comment for this command
another_cmd = -3.0 # Another comment.
array_1d(0) = 2,4,-3 // A comment
! *** whatever
! another comment line
! comment out this command: cmd(1) = true, false, .true.
```

Single line comments can, of course, be nested as in:

```
cmd = True ! True or true or // .true. or ! tRue are all ok.
```

Everything from the first "!" to the end of the line is treated as a comment and ignored.

The multi-line comment characters are "/*" to start the comment and "*/" to end the comment (same as in C/C++). Everything between these, inclusive, is treated as a comment and ignored. Some examples follow:

```
some_cmd = 5.0 /* Some comment for this command */
array_1d(0) = /* Embedded comment */ 2,4,-3
/* Several lines of comments.
* line 2 comment
line 3 comment
*/ // End comment block.
```

This is another example of a multi-line comment, the *cmdml* command gets processed normally:

```
/* This is a
* multi-line
* comment */ cmdml(1) = 14.6, 17.8, 10.9, 1.e19
```

Single line comments are processed first, then multi-line comments are processed. Thus the following will produce a fatal error about unbalanced comment characters:

```
/* This is a
* multi-line
* comment ! */ cmdml(1) = 14.6, 17.8, 10.9, 1.e19
```

The "!" character in line 3 is processed first and removes the "*/" comment character (and the *cmdml* command). Then the multi-line comments are processed and the "/*" comment character in line 1 has no matching "*/" and produces a fatal error.

Multi-line comments can be nested, as in:

```
array_1d(0) = /* Embedded /* nest 1 /* nest 2 */ */ comment */ 2,4,-3
```

The basic rule is that every "/*" starting comment delimiter must have an associated "*/" ending comment delimiter. Here is another example of multi-line comment nesting:

```
/* This is a
* multi-line
/*
* comment cmdml(1) = 14.6, 17.8, 10.9, 1.e19
*/
* with
* nesting. */
```

3.4 Variables

Variables (both scalars and arrays) can be defined and used in the user input file. Variable names always begin with the "$" character. An example of defining a scalar variable and using it follows:

```
$rho1 = 1.34
acmd = $rho1
```

PowerParser would set the variable associated with "acmd" to 1.34. Variables can also be used in math expressions (see Math Operations Section for a discussion of math expressions) for example:

```
$angle = 35.7
$PI = 3.1415926535897932
math_result12 = (sin($angle*$PI/180.)**2 + &
cos($angle*$PI/180.)**2)
```

The math expression evaluates to 1.0.

In addition to scalar varables, multi-dimensional variable arrays are allowed. In the following example, a 1D variable is defined and used in a math expression.

```
$var1d(0) = 2. 3. 4.
var1d_res = ($var1d(2) * 10.)
```

The first element of the $var1d array is 2., the second element is 3., and the third element is 4. The second element of $var1d, 3., is used in the math expression, it is multiplied by 10., with the result of 30. Thus *PowerParser* would set the variable associated with var1d_res to 30.

Indices in variable arrays always start from 0.

Following is an example of a 2d variable definition and its use:

```
$var2d dimension(3,:)
$var2d(0,0) = 11. 21. 31. 12. 22. 32. 13. 23. 33.
var2d_res = ($var2d(2,1) * 10.)
```

Multi-dimensional variable arrays have a problem in that the parser needs to be told the maximum value of each dimension, except for the very last one. In the above example, the dimension command is used to tell the parser that the maximum of the first dimension for the $var2d variable array is 3. The second and final dimension is set to ":" indicating to the parser that it will be set as the user inputs values.

These rules follows the Fortran convention where the first index of an array varies fastest and the last index varies slowest.

The second line in the above example actually sets the $var2d values, the first 3 values correspond to elements (0,0), (1,0), and (2,0). The next 3 values correspond to elements (0,1), (1,1), and (2,1) and so on.

The last line of the above example multiplies element (2,1) of the $var2d array with 10. Since element (2,1) is 32., the result is 320. *PowerParser* will thus set the variable associated with var2d_res to 320.

The need for the dimension statment for multi-dimensional variables is illustrated in the following example. Suppose the user inputs the following command:

```
$var2d(0,0) = 11. 21. 31. 12. 22. 32.
```

If the maximum of the first dimension is 3, then the parser will set the elements as:

```
$var2d(0,0) = 11.
$var2d(1,0) = 21.
$var2d(2,0) = 31.
$var2d(0,1) = 12.
```

```
$var2d(1,1) = 22.
$var2d(2,1) = 32.
```

But if the maximum of the first dimension were 2, then it would be equivalent to:

```
$var2d(0,0) = 11.
$var2d(1,0) = 21.
$var2d(0,1) = 31.
$var2d(1,1) = 12.
$var2d(0,2) = 22.
$var2d(1,2) = 32.
```

A reference to element (0,0) in the first scheme would be 22., but it would be 12. in the second way. Thus to avoid this ambiguity, the parser must be told the maximum values of all dimensions except the last, before the variable is defined.

Any number of dimensions is allowed in variable arrays. The following is an example of the definition and use of an 8 dimensional variable (admittedly, this is a somewhat ridiculous example but it does work).

```
$var8d dimension(3,2,1,2,2,3,2,:)
$var8d(0,0,0,0,0,0,0,0) = &
00000000 10000000 20000000 &
01000000 11000000 21000000 &
00010000 10010000 20010000 &
01010000 11010000 21010000 &
00001000 10001000 20001000 &
01001000 11001000 21001000 &
00011000 10011000 20011000 &
01011000 11011000 21011000 &
00000100 10000100 20000100 &
01000100 11000100 21000100 &
00010100 10010100 20010100 &
01010100 11010100 21010100 &
00001100 10001100 20001100 &
01001100 11001100 21001100 &
00011100 10011100 20011100 &
01011100 11011100 21011100 &
00000200 10000200 20000200 &
01000200 11000200 21000200 &
00010200 10010200 20010200 &
01010200 11010200 21010200 &
00001200 10001200 20001200 &
01001200 11001200 21001200 &
00011200 10011200 20011200 &
01011200 11011200 21011200 &
00000010 10000010 20000010 &
01000010 11000010 21000010 &
00010010 10010010 20010010 &
01010010 11010010 21010010 &
00001010 10001010 20001010 &
01001010 11001010 21001010 &
00011010 10011010 20011010 &
01011010 11011010 21011010 &
00000110 10000110 20000110 &
01000110 11000110 21000110 &
00010110 10010110 20010110 &
01010110 11010110 21010110 &
00001110 10001110 20001110 &
01001110 11001110 21001110 &
```

```
00011110 10011110 20011110 &
01011110 11011110 21011110 &
00000210 10000210 20000210 &
01000210 11000210 21000210 &
00010210 10010210 20010210 &
01010210 11010210 21010210 &
00001210 10001210 20001210 &
01001210 11001210 21001210 &
00011210 10011210 20011210 &
01011210 11011210 21011210

var8d_cmd = ($var8d(2,1,0,1,0,1,1,0))
```

PowerParser will set the variable associated with var8d_cmd to 21010110.

Variable arrays can be used outside of math expressions, the previous example could also have been written as:

```
var8d_cmd = $var8d(2,1,0,1,0,1,1,0)
```

In addition to numerical values, variables can also use logical and character values. The following is an example of the use of logical values.

```
$log1d(0) = true false false true true false
log1d_cmd = (.not.($log1d(4) .and. $log1d(5)))
```

PowerParser will set the variable associated with log1d_cmd to .true.

This is an example of a 3d variable array using character strings.

```
$vchar3d dimension(2,1,:)
$vchar3d(0,0,0) = Turn off your computer Coker "," use &
                  the force
vchar3d_cmd = ($vchar3d(0,0,1))
```

PowerParser will set the variable associated with vchar3d_cmd to “use”. Note that in the above example, the comma between the words Coker and use had to be put in quotes, otherwise the parser would remove it.

Variable arrays can also have mixed value types. In the following example the variable array vnc has character, real, logical, and integer values.

```
$vnc(0) = Turn 1.0e19 true -3
vnc_cmd = (2*$vnc(1))
```

PowerParser will set the variable associated with vnc_cmd to 2.0e19.

Variables can have optional user supplied descriptions associated with them. As seen above, the pre-defined variables all have descriptions. Users can set descriptions for user defined variables with the thevariable_description command, for example:

```
variable_description $radius radiusradius of something (cm)
```

The second word is the name of the variable, the third word is the description, usually a phrase in quotes.

The user can print a list of variables at any point in the input file with the “parser_list_variables” command, for details of this command see ‘Debugging User Input‘_.

3.5 Intrinsic Functions

Several intrinsic functions are provided and can be used in the user input file. Intrinsic functions can only be used in math expressions. An example of the use of an intrinsic function is:

```
f01 = (exp(log(exp(log(4.58)))))
```

The math expression evaluates to 4.57999999999998.

Another example is:

```
f02 = (acos(0.5)*180./$pi)
```

which evaluates to 60.0000000000001.

The above functions are called "real" functions in the parser, meaning they take double (C++ double type) arguments and return a double result. The parser also has string functions which take string arguments and return string or integer results. For example, the strlen function can be used as:

> strlen_cmd01 = (strlen("I felt a great disturbance in the Force")) $strlen_var = 1.0e14 strlen_cmd02 = (strlen($strlen_var))

PowerParser will set the variable associated with strlen_cmd01 to 39 and will set the variable associated with strlen_cmd02 to 6.

An example of using the strcat function is:

```
strcat_cmd01 = (strcat(strcat("Obi", "-Wan"), " Kenobi"))
```

PowerParser will set the variable associated with strcat_cmd01 to "Obi-Wan Kenobi". Another example of using strcat is:

```
$strcat_var = "Obi"
do $i = 1,3
   $strcat_var = (strcat($strcat_var, $i))
enddo
strcat_cmd02 = $strcat_var
```

PowerParser will set the variable associated with strcat_cmd02 to Obi123.

The strerase function erases characters from a string. This function takes 3 arguments, the first is the string to erase characters from, the second is the starting position in the string for the erase, and the third is the ending position for the erase. Both the starting and ending positions start from 1 (Fortran style). For the following example:

```
strerase_cmd01 = (strerase("The Force", 1, 4))
$strerase_var02 = 1.3e14
strerase_cmd02 = (strerase($strerase_var02, 3, 3))
```

PowerParser will set the variables associated with strerase_cmd01 to "Forc"e and strerase_cmd02 to 1.e14.

The strinsert function inserts characters into a string. This function takes 3 arguments, the first is the string to insert into, the second is the insert position and the third is the string to be inserted. To insert at the very end of the string use the number of characters in the string plus 1 for the insert position. You could alternatively use the strcat function to append one string to another. The use of the strinsert function is illustrated in the following example:

```
strinsert_cmd01 = (strinsert(strinsert("The", 1, "Use "), &
                   8, " Force"))
```

PowerParser will set the variable associated with strinsert_cmd01 to "Use The Force".

The strsubstr function extracts a substring from the string. This function takes 3 arguments, the first is the string to extract from, the second is the position in the string to start the extraction (starting from 1, Fortran style), and the third argument is the number of characters to extract. The use of strsubstr is shown in the following example:

```
strsubstr_cmd01 = (strsubstr("Use The Force", 5, 3))
```

PowerParser will set the variable associated with strsubstr_cmd01 to "The".

The strtrim function removes trailing whitespace from the string. Whitespace in this context is defined as spaces and tabs. This takes one argument, the string to trim. The following example shows the use of this function:

```
strtrim_cmd01 = (strcat(strtrim("Use The Force "), &
                 ", Scott"))
```

PowerParser will set the variable associated with strtrim_cmd01 to "Use The Force, Scott". The currently (as this manual is being written) available intrinsic functions are:

```
********** Debugging: list function names
   Function name  nargs   type                 Description
   -------------  -----  ------    -------------------------------
          acos      1      real    arccosine, radians, arg -1 to 1
          asin      1      real      arcsine, radians, arg -1 to 1
          atan      1      real        arctangent, returns radians
         atan2      2      real                arctangent, 2 args
          ceil      1      real     round up (smallest int >= arg)
           cos      1      real             cosine, arg in radians
          cosh      1      real                  hyperbolic cosine
           exp      1      real                        exponential
          fabs      1      real           absolute value of a real
         floor      1      real    round down (largest int <= arg)
          fmod      2      real             remainder of arg1/arg2
           log      1      real               log to base e, arg>0
         log10      1      real              log to base 10, arg>0
           max      2      real     return the greater of two args
           min      2      real      return the lesser of two args
           pow      2      real          arg1 raised to arg2 power
           sin      1      real               sine, arg in radians
          sinh      1      real                    hyperbolic sine
          sqrt      1      real             square root (arg >= 0)
        strcat      2    string            concatenate two strings
     strerase      3    string            erase chars from string
    strinsert      3    string           insert chars into string
       strlen      1    string             number of chars in arg
    strsubstr      3    string                      get sub string
      strtrim      1    string         remove trailing whitespace
          tan      1      real            tangent, arg in radians
         tanh      1      real                 hyperbolic tangent
********** Debugging END: list function names
```

To get the latest list of intrinsic functions, put the command "parser_list_functions" anywhere in your input file. See 'Debugging User Input'_ for a description of this command.

3.6 Math Operations

Math operations are allowed in certain places in the user input file. Math operations are constructed using the following:

```
parentheses (any level of nesting allowed)
variables (for example $radius)
arithmetic operators (**, *, /, +, -)
relational operators (.gt., .ge., .lt., .le., .eq., .ne.)
logical operators (.not., .and., .or.)
intrinsic functions (cos, sin, exp, ...)
```

Some math examples are shown in Table 3.1.

Expression	Result
(1+2*3)	7
(4. + 4.*(sin(30.*$pi/180.) * cos(60.*$pi/180.)))	5
(.not. false)	true

Table 3.1: A few math results

Math operations are allowed inside parentheses, i.e. anything inside a set of parentheses is sent to the math evaluator. An example of math being done is:

```
(4 - -5)
```

The results of this math operation is 9. The math operation is allowed in command lines and variable assignment lines. For example:

```
delta_x = (50. / 5000.)
```

When the math expression is evaluated, this results in:

```
delta_x = 0.01
```

When *PowerParser* scans the compiled input file for "delta_x", it will set the associated variable to 0.01.

An example of an assignment command using math is:

```
$r = 1.2
$volume = (4*$pi*$r**3/3)
```

( )	Highest
++ - - (postfix only)	
**	
* /	
+ -	
.gt. .ge. .lt. .le. .eq. .ne.	
.not.	
.and.	
.or.	Lowest

Table 3.2: Table of operator precedence levels. Operators with higher precedence are done first. Operators within a level are done from left to right as they are encountered.

which sets the variable $volume to about 7.238. This example also illustrates operator precedence. The exponentiation (**) is done first, then the multiplication (*) and division (/) is done from left to right. Table 5.2 shows the operator precedence. As another example, consider the following expression:

```
math_result17 = (5.gt.4.or.10.gt.20.and.false)
```

First this tests the parser's ability to parse relational and logical operators (.gt., .and., etc) as separate words, i.e. spaces are not needed around the operators. Second, this demonstrates precedence order of operations. The relational operators (.gt.) are processed first, giving:

```
true .or. false .and. false
```

Then the .and. logical operator is processed giving:

```
true .or. false
```

Finally the .or. logical operator is processed giving the final result of true. If the .or. operator had been processed first, the result would have been false. We can force this by adding additional parentheses:

```
math_result18 = ((5.gt.4.or.10.gt.20).and.false)
```

This gives false as a final result.

The ++ and - - operators (from C/C++) follow a variable and after the variable is used will increment (for ++) or decrement (for - -) the value of the variable by 1. This is shown in the following example:

```
$ppmm_v1 = 5
ppmm_cmd01 = ($ppmm_v1++) ! Should be 5
ppmm_cmd02 = ($ppmm_v1) ! Should be 6
ppmm_cmd03 = ($ppmm_v1--) ! Should be 6
ppmm_cmd04 = ($ppmm_v1) ! Should be 5
```

Variable $ppmm_v1 in this example starts as 5, in line 2 it is used frst, then incremented, so ppmm_cmd01 is 5. Since the variable was incremented in line 2, its value in line 3 is 6 and thus ppmm_cmd02 is 6. In line 4 the value of $ppmm_v1 is still 6 so ppmm_cmd03 is 6, then the variable gets decremented and its value is 5. In the fnal line of the example, the variable still has a value of 5 and ppmm_cmd04 is 5.

The main reason for implementing the ++ and - - operators was to produce more compact input files. Consider the following example:

```
$r = 1
matreg($r) = $al_can
$r = ($r + 1)
matreg($r) = $cu_sphere
$r = ($r + 1)
matreg($r) = $neon_tube
```

This can be rewritten using the ++ operator as:

```
$r = 1
matreg($r++) = $al_can
matreg($r++) = $cu_sphere
matreg($r) = $neon_tube
```

Another example of using the ++ operator follows:

```
$ppmm_v1 = 5
ppmm_cmd05 = ($ppmm_v1++ + $ppmm_v1) ! Should be 11
```

The first occurrence of $ppmm_v1 is used and replaced by 5, then $ppmm_v1 is incremented by 1 giving it a value of 6. The line becomes the following:

```
ppmm_cmd05 = (5 + $ppmm_v1) ! Should be 11
```

Since $ppmm_v1 is now 6, the line becomes:

```
ppmm_cmd05 = (5 + 6) ! Should be 11
```

and the final result for ppmm_cmd05 is 11.

The ++ and - - operators can follow a variable array reference as in the following example:

```
$ppmm_v2(1) = 7 19 11
ppmm_cmd06 = ($ppmm_v2(2)++) ! Should be 19
ppmm_cmd07 = ($ppmm_v2(2)) ! Should be 20
```

The ++ and - - operators can only follow a variable or a variable array reference. The following fails because the ++ operator follows a number rather than a variable.

```
ppmm_cmd08 = (2++) ! Should fail.
```

The following is also not allowed because the ++ operator is not immediately following the variable.

```
ppmm_cmd08 = ($ppmm_v1)++ ! Should fail.
```

In the previous examples, the ++ and - - operators have been inside math expressions. But they can also appear outside math expressions as in the following example:

```
ppmm_cmd08 = $ppmm_v1++ ! Should be 6
ppmm_cmd09 = $ppmm_v1 ! Should be 7
ppmm_cmd10 = $ppmm_v2(2)-- ! Should be 20
ppmm_cmd11 = $ppmm_v2(2) ! Should be 19
```

The ++ and – operators can also apply to real variables as shown in the following example:

```
$ppmm_v1 = 17.356
ppmm_cmd12 = $ppmm_v1++ ! Should be 17.356
ppmm_cmd13 = $ppmm_v1 ! Should be 18.356
```

Following is another example of using the ++ operator:

```
// $ppmm_v1 = ($ppmm_v1++) $ppmm_v1 is 5
// $ppmm_v1 = (5++) The var gets replaced by 5
// $ppmm_v1 = (5) ++ is done, $ppmm_v1 is now 6
// $ppmm_v1 = 5 The assignment is done, $ppmm_v1 is 5 again
// So $ppmm_v1 ends up unchanged.
$ppmm_v1 = 5
$ppmm_v1 = ($ppmm_v1++)
ppmm_cmd14 = $ppmm_v1 ! Should be 5
```

The following is an example of using the ++ operator with a 2d array:

```
$ppmm_v2d dimension(3,:)
$ppmm_v2d(1,1) = 11. 21. 31. 12. 22. 32. 13. 23. 33.
ppmm_cmd15 = ($ppmm_v2d(3,2)++*-10) ! Should be -320
ppmm_cmd16 = $ppmm_v2d(3,2) ! Should be 33
```

More examples of math results are given in Table 3.3.

Expression	Result
(1.1 + 2.7*2.0/2)	3.8
(fmod(15,4)+3)	6
(4*-5)	-20
(13+(4*(15/3+4))) |	49
(sin(30.*$pi/180.)**2 + cos(30.*$pi/180.)**2)	1

Table 3.3: More examples of math results

3.7 Integer Arithmetic

All arithmetic in the math operations is done in double precision. Thus (4/8) and (4./8.) are both evaluated to be 0.5. Integer arithmetic, where (4/8) would be 0, can be effectively accomplished by using the modulus intrinsic function, fmod, and the ceil and floor intrinsic functions. For example, to find the remainder of 15 divided by 4, use the modulus intrinsic function:

```
iarith_cmd01 = (fmod(15,4))
```

PowerParser will set the variable associated with iarith_cmd01 to 3. To round down, use the floor function:

```
iarith_cmd02 = (floor(15/4))
```

PowerParser will set the variable associated with iarith_cmd02 to 3. To round up, use the ceil function:

```
iarith_cmd03 = (ceil(15/4))
```

PowerParser will set the variable associated with iarith_cmd03 to 4.

3.8 If Statements

If statements are allowed in user input files. Two types of if statements are available, the single line if and the multi-block if. Following is an example of the single line if:

```
$dimension = 2
if ($dimension .eq. 2) delta_y_cmd01 = 1.0
if ($dimension .eq. 1) delta_y_cmd01 = 2.0
```

Since $dimension is set to 2, the rst if evaluates to true and everything following the closing parentheses is processed as a normal command:

```
delta_y_cmd01 = 1.0
```

PowerParser will set the variable associated with delta_y_cmd01 to 1.0. The second if is ignored since it evaluates to false. Variables can be set by single line if statements as in the following example:

```
$delta_y = 1.0
$dimension = 2
if ($dimension .eq. 1) $delta_y = 5.0
delta_y_cmd02 = $delta_y
if ($dimension .eq. 2) $delta_y = 3.0
delta_y_cmd03 = $delta_y
```

PowerParser will set the variable associated with delta_y_cmd02 to 1.0 and set the variable associated with delta_y_cmd03 to 3.0. Another example of a single line if using arrays follows:

```
$dimension = 2
$ifarray(1) = 4*0
if ($dimension .eq. 2) $ifarray(1) = 1, 2, 3, 4
delta_y_cmd04 = $ifarray(3)
```

$ifarray is a 4 element array initialized to 0. The if statement evaluates to true and sets the array to 1 through 4. *PowerParser* will set the variable associated with delta_y_cmd04 to 3.

Any kind of math expression is allowed in the if conditional, i.e. inside the parentheses, with the rule that the math expression must evaluate to either true or false.

Multi-block if statements using the if, elseif, else, and endif commands are allowed. A simple if...endif block is shown in the following example:

```
$dimension = 2
if ($dimension .eq. 2) then
   $delta_y = 0.1
endif
delta_y_cmd05 = $delta_y
```

PowerParser will set the variable associated with delta_y_cmd05 to 0.1.

Multi-block if statements can be nested, to any level, as in the following example:

```
$dimension = 2
$multid = true
if ($multid) then
   if ($dimension .eq. 2) then
      $delta_y = 0.2
   endif
endif
delta_y_cmd06 = $delta_y
```

PowerParser will set the variable associated with delta_y_cmd06 to 0.2.

The following example shows the use of the else statement in a block if:

```
$dimension = 2
if ($dimension .eq. 1) then
   $delta_y = 0.13
else
   $delta_y = 0.26
endif
delta_y_cmd07 = $delta_y
```

PowerParser will set the variable associated with delta_y_cmd07 to 0.26.

An example using the elseif command follows:

```
$dimension = 2
$problem_name = "test01"
if ($problem_name .eq. "test01") then
   if ($dimension .eq. 1) then
      $delta_x = 0.23
   else if ($dimension .eq. 2) then
      $delta_x = 1.48
   elseif ($dimension .eq. 3) then
      $delta_x = 2.56
   elseif ($dimension .eq. 4) then
      $delta_x = 5.6
   else
      $delta_x = 9.1
   endif
end if
delta_x_cmd01 = $delta_x
```

PowerParser will set the variable associated with delta_x_cmd01 to 1.48. Note that endif can be written either as "endif" or as "end if", similarly elseif can be written as either "elseif" or "else if".

Following is a more complicated example:

```
$dimension = 3
$problem_name = "test01"
$run_number = 12
$popt01 = "opt01"
if ($problem_name .eq. "test01") then
   if ($run_number .eq. 12) then
      if ($dimension .eq. 1) then
         $delta_x = 0.23
      else if ($dimension .eq. 2) then
         $delta_x = 1.48
      elseif ($dimension .eq. 3) then
         if ($popt01 .eq. "opt03") then
            $delta_x = 2.96
         elseif ($popt01 .eq. "opt02") then
            $delta_x = 21.96
         else
            $delta_x = 43.56
         endif
      elseif ($dimension .eq. 4) then
         $delta_x = 5.6
      else
         $delta_x = 9.1
      endif
   end if
end if
delta_x_cmd02 = $delta_x
```

PowerParser will set the variable associated with delta_x_cmd02 to 43.56.

3.9 Do Loops

Do loops are allowed in user input files. Here is an example of a simple do loop:

```
$sum = 0
do $i=1,10
   $sum = ($sum + 1)
enddo
do_sum_cmd01 = $sum
```

The loop variable is "$i" and goes from 1 to 10 with a default step size of 1. The $sum variable is accumulated and will end up being 10. *PowerParser* will set the variable associated with do_sum_cmd01 to 10.

Note that the loop variable name begins with a "$". All variable names in the user input file must begin with a dollar sign, "$".

Do loops can be nested as in the following example:

```
$sum = 0
do $i=1,10
   do $j=1,10
      do $k=1,10
         $sum = ($sum + 1)
      enddo
   enddo
enddo
do_sum_cmd02 = $sum
```

PowerParser will set the variable associated with do_sum_cmd02 to 1000.

The exit statement can be used to break out of a loop, this is shown in the following example:

```
$sum = 0
do $i=1,10
   do $j=1,10
      do $k=1,10
         if ($k .eq. 3) exit
         $sum = ($sum + 1)
      enddo
   enddo
enddo
do_sum_cmd03 = $sum
```

PowerParser will set the variable associated with do_sum_cmd03 to 200.

The cycle command cycles to the next iteration of the loop as shown in the following example:

```
$sum = 0
do $i=1,10
   do $j=1,10
      do $k=1,10
         if ($k .eq. 3) cycle
         $sum = ($sum + 1)
      enddo
   enddo
enddo
do_sum_cmd04 = $sum
```

PowerParser will set the variable associated with do_sum_cmd04 to 900.

A step size can be specifed as the last parameter of the do statement as shown in the following example:

```
$sum = 0
do $i=1,10,2
   do $j=1,10,3
      do $k=1,10,4
         if ($k .eq. 5) cycle
         $sum = ($sum + 1)
      enddo
   enddo
enddo
do_sum_cmd06 = $sum
```

PowerParser will set the variable associated with do_sum_cmd06 to 40.

The start, stop, and step values specified in the do statement can be math expressions. They must evaluate to integers and as for all math expressions, they must be enclosed in parentheses. This is shown in the following example:

```
$sum = 0
do $i = (sin(0.3)**2+cos(0.3)**2), (8/4*3+13)
   $sum = ($sum + $i)
enddo
do_sum_cmd05 = $sum
```

PowerParser will set the variable associated with do_sum_cmd05 to 190.

The start, stop, and step values specified in the do statement can be variables as in the following example:

```
$istart = 1
$iend = 10
do $i=$istart,$iend
```

This is equivalent to the following:

```
do $i=1,10
```

This is also equivalent to the following:

```
$istart_array(1) = 3, 2, 1, 4
do $i=$istart_array(3),10
```

The start, stop, and step values specified in the do statement can be negative as in the following example:

```
$sum = 0
$do_stop = 3
do $i=3,-$do_stop,-1
   $sum = ($sum + 1)
enddo
do_sum_cmd07 = $sum
```

PowerParser will set the variable associated with do_sum_cmd07 to 7.

3.10 Subroutines

Subroutines are allowed in user input files. Here is an example of a simple subroutine:

```
subroutine test1
   $test1_var = 3
end subroutine
```

All it does is set the global variable, $test1_var, to 3. The call to this subroutine is:

```
call test1
sub_cmd01 = $test1_var
```

PowerParser will set the variable associated with sub_cmd01 to 3.

Subroutines can appear anywhere in the user input file, except that subroutines cannot be inside do loops. Subroutines can be called from inside do loops, they just cannot be defined inside the do loop.

Subroutines can be nested, i.e. subroutines can call each other, as shown in the following example:

```
subroutine test2
   $test2_var = 3
   call test3
end subroutine

subroutine test4
   $test2_var = 9
end subroutine

subroutine test3
   $test2_var = 6
   call test4
end subroutine
```

Subroutine test2 calls test3 which calls test4. Note that test4 appears before test3. This, of course, is allowed. The call to test2 is:

```
call test2
sub_cmd02 = $test2_var
```

PowerParser will set the variable associated with sub_cmd02 to 9.

Subroutines are allowed to have arguments, the following subroutine has 4 arguments:

```
subroutine twa1($arg1, $arg2, $arg3, $arg4)
   $twa1_var1 = $arg1
   $twa1_var2 = $arg2
   $arg1 = 52
   //$arg2 = 52 // Uncomment to test error of changing
   // a fixed argument.
   $arg3(2) = 16
   $arg3(3) = 7
   $twa1_var4 = $arg4
end subroutine
```

The corresponding call statement for this subroutine is:

```
$targ1 = 3
$targ1_arr1d(0) = 4*300
call twa1($targ1, (4*(sin(0.5)**2 + cos(0.5)**2)), $targ1_arr1d, &
          $targ1_arr1d(3))
sub_cmd03 = $twa1_var1
sub_cmd04 = $twa1_var2
sub_cmd05 = $targ1
sub_cmd06(0) = $targ1_arr1d(1) $targ1_arr1d(2) &
               $targ1_arr1d(3) $targ1_arr1d(4)
sub_cmd07 = $twa1_var4
```

The first argument in the call statement is a variable, $targ1, the second argument is a math expression which evaluates to 4, the third argument is a 1d array which has 4 elements, the fourth argument is one element of the array and has the value of 300.

Every argument in the call statement is sent to the math evaluator before being sent to the subroutine, thus the intermediate call statement for this example, after math evaluation is:

```
call twa1($targ1, 4, $targ1_arr1d, 300)
```

Arguments 1 and 3 are passed in as variables which can be modified by the subroutine, arguments 2 and 4 are fixed numbers which cannot be changed by the subroutine.

The first two lines in the example subroutine twa1 above set the global variables $twa1_var1 and $twa1_var2. *PowerParser* will set the variables associated with sub_cmd03 and sub_cmd04 to these variable values, 3 and 4. The third line changes the value of $arg1 to 52, which changes the value of the calling argument, $targ1, to 52. *PowerParser* will set the variable associated with sub_cmd05 to 52. The fourth line, if uncommented, would attempt to set the fixed value of 4, calling argument 2, to 52, this is not allowed and would generate a fatal error. Lines 6 and 7 of the subroutine set the array elements 2 and 3 to 16 and 7 respectively. The calling argument, $targ1_arr1d, has 4 array elements all initialized to 300, the subroutine changes elements 2 and 3 to 16 and 7 respectively. *PowerParser* will set the variable array associated with sub_cmd06 to 300, 16, 7, 300. The last line of the subroutine sets the global variable $twa1_var4 to $arg4 which was passed in as the fixed number 300. *PowerParser* will set the variable associated with sub_cmd07 to 300.

Return statements are allowed in subroutines as in the following example:

```
subroutine twret1($arg1, $arg2)
   $arg1 = 52
   return
   $arg2 = 16
end subroutine
```

The first line sets the value of the first argument to 52. The second line returns control to the calling routine. The third line is never processed.

A more complicated example of using a return statement is shown in the following example:

```
subroutine twret2($arg1)
   $arg1 = 0
   $retif01 = true
   do $i=1,10,2
      do $j=1,10,3
         if ($retif01 .eq. true) then
            do $k=1,10,4
               if ($j .eq. 7) return
               $arg1 = ($arg1 + 1)
            enddo
         endif
      enddo
   enddo
end subroutine
```

The associated call to this subroutine is:

```
$targ1 = 0
call twret2($targ1)
sub_cmd10 = $targ1
```

PowerParser will set the variable associated with sub_cmd10 to 6.

3.11 Include Files

Other files can be included in the user input file with the use of the "include" command, which has the form:

```
include "filename" "alternate_filename1" "alternate_filename2" ...
```

For example, suppose the user input file is called "test.in" and we have another file called "test_inc1.in" that we wish to include in "test.in". The include file, "test_inc1.in", is very simple and only has one line in it:

```
$tinc01 = 3.0
```

It just sets variable $tinc01 to 3.0. The user input file, "test.in" contains the following two lines:

```
include "test_inc1.in"
inc_cmd01 = $tinc01
```

PowerParser will set the variable associated with inc_cmd01 to 3.0.

If the file name is a simple, one word file name, then it does not need to be in quotes, but if it is more complicated such as having a path prepended to it, then it will need to be in quotes. It is perhaps good practice to always use quotes.

Include statements can be nested to any level, consider the following two files for example, file test_inc2.in

```
$tinc02 = 5.0
include "test_inc3.in"
$tinc03 = 7.0
```

and file test_inc3.in

```
$tinc04 = -19.0
```

The user input has the following lines:

```
include "test_inc2.in"
inc_cmd02 = $tinc02
inc_cmd03 = $tinc03
inc_cmd04 = $tinc04
```

thus it includes file letest_inc2.in which in turn includes file letest_inc3.in. *PowerParser* will set the variables associated with inc_cmd02 to 5.0, inc_cmd03 to 7.0, and inc_cmd04 to -19.0.

A problem with the include command is that it is not always portable. The specified include file may exist on one system, but may not exist on another system. Or perhaps it has a different name, or is located in a different place. To handle this, the include command allows alternate filenames. For example:

```
include "file1" "/dir2/file2" "../../file3"
```

The parser checks to see if file1 exists, if it does then the parser opens it and processes it. If file1 does not exist, then the parser checks file2. If that does not exist, then it checks file3. Any number of file names may be put on the include line. The file names must be blank delimited (not comma delimited) and they must all be on one line, no continuation characters are allowed on include lines.

Note that the include command is processed very early in the input file processing phase thus features such as variables, math expressions, etc, are not allowed in include commands.

The include can be commented out with single line comments, for example the following include will not be done:

```
// include "test_inc1.in"
```

Multi-line comments, however, as in the following example, will not comment out the include. Of course, everything that got included in this example will be commented out.

```
/*
include "test_inc1.in"
*/
```

3.12 Execution Line Arguments

When a program is executed, the usual execute line is, for example:

```
./myProgram test.in
```

where "myProgram" is the name of the executable and test.in is the name of the input file. When running in parallel, mpirun would also be used and the number of processors would be specified.

PowerParser allows additional execute line arguments after the name of the input file. These additional arguments use the "-v" keyword and specify parser variables and values. The "-v" arguments are by default parsed first and effectively added to the top of the input file. Thus their associated variables can be used to control aspects of the input file. See below, however, for a description of the "put_exe_args_here" command which allows the user to put the execution line arguments anywhere in the input file.

For example, suppose the execute line is:

```
./myProgram test.in -v dophysopt=true
```

The name of the variable is "dophysopt" and its value is true. Variables normally always begin with "$", but the unix shell will not allow that on the execute line, thus the parser adds the "$" when it is added to the input file.

Suppose the input file has the following lines:

```
if ($dophysopt .eq. true) then
   physics_cmd1 = 3.0
   physics_cmd2 = 6.0
endif
```

If the execute line defines "dophysopt" to be true, then the physics commands are processed. If the execute line defines "dophysopt" to be false, then the physics commands are not processed.

If the execute line does not define "dophysopt" at all, then a fatal error would be generated when attempting to use an undefined variable. Usually, this is not what the user wants, the user would like the input file to work in some default manner when no execute line arguments are specified. One way to accomplish this is with the use of the "put_exe_args_here" command which can be put anywhere in the input file. If this command is present in the input file, then it is replaced with the execute line arguments and the execute line arguments are not put at the top of the file. The "put_exe_args_here" command should only appear once (if at all) in the input file. If the "put_exe_args_here" command is not present in the input file, then the execute line arguments are put at the top of the input file.

Thus the previous example would be better written as:

```
$dophysopt = false
put_exe_args_here
if ($dophysopt .eq. true) then
   physics_cmd1 = 3.0
   physics_cmd2 = 6.0
endif
```

This defaults $dophysopt to false which then can be overridden with any execution line arguments. Then if the user does not specify dophysopt on the execute line, the program will run fine and use the $dophysopt default of false.

Another way (and perhaps better way) to handle execution line variable defaults is to use the parser intrinsic "defined" function which returns true if the variable has been previously defined and otherwise returns false. Thus the previous example could also be written as:

```
if (.not. defined("$dophysopt")) then
   $dophysopt = false
endif
if ($dophysopt .eq. true) then
   physics_cmd1 = 3.0
   physics_cmd2 = 6.0
endif
```

Any number of "-v" arguments are allowed, for example:

```
./myProgram test.in -v dophysopt=true -v r1=5 -v r2=3e19
```

This would be equivalent to having the following three lines at the top of the input file (or whereever the "put_exe_args_here" command is)

```
$dophysopt=true
$r1=5
$r2=3e19
```

You can now specify "-l" options on the command line (as well as "-v"). The "-l" means to insert the line directly into the input file. The "-v" is used for a variable definition.

```
Example:
   mpirun -np 1 ./myProgram foo.in -l pname=foo -v ncmax_var=100
Input File Original:
   [...]
   put_exe_args_here
   ncmax = $ncmax_var
   [...]
Effective Input File:
   [...]
   pname=foo // inserted the -l line directly
   $ncmax_var=100 // prepended "$" to ncmax_var since used the "-v" option
   ncmax = $ncmax_var
   [...]
```

NOTE: Since the history of what "-v" and "-l" flags is only saved if you save your output and log files, these options should be used sparingly...and document them well so you can reproduce your runs.

3.13 Checking Input

A new execute line argument, "-check_input", has been added. This reads the input file, issues any warning or fatal error messages and then exits. It allocates very little memory and thus can be run with one processor and even run on a front-end machine.

An example of using the -check_input execution line argument follows:

```
./myProgram test.in -check_input
```

This feature is mostly intended to help with calculations using many processors, especially when running in batch mode. The user input can then be easily checked using one processor prior to submitting the run.

3.14 Duplicate Input

PowerParser originally allowed duplicate commands in the user input. When duplicate input is found, *PowerParser* uses the last instance of the duplicate input. In the interest of better error checking, this behavior has been modified somewhat.

The *PowerParser* distinguishes between scalar commands and array commands. Handling duplicate input is different for the two types of commands. An example of a scalar command is:

```
do_rad = false
```

An example of an array command is:

```
mults(0,0) = 0. 0. 1. 5. 6. 9.
```

If a scalar command is in the input file more than once, then the parser issues a warning to the user. The intent is to simply remind the user that duplicate commands exist and is the setting for the last one really what the user wants. For example, suppose the user input includes the following:

```
...
do_phys = false
...
do_phys = true
...
do_phys = false
...
```

In this example, the user has 3 occurrences of the do_rad command at various places in the input. The parser will issue something like the following warning for this example:

```
********** WARNING: Duplicate Scalar Commands Found in User Input File
The following commands appear more than once in the user input file.
The last instance of the command will be used.
Is this what you want??

              Line
  Filename    Number                  Command
------------  ------  ---------------------------------------
     test.in       4  do_phys = false
     test.in      28  do_phys = true
     test.in     110  do_phys = false
```

The hope is that the user will see this warning in the screen output and be cognizant of what setting for do_phys the code is actually using.

If an array value is in the input file more than once, this is now flagged as a warning and a warning message will be printed. There is a command to turn this into a fatal error to force the user to check the input. Consider for example the two input lines:

```
mults(0,0) = 0. 0. 1. 5. 6. 9.
mults(5,0) = 11. 13. 22. 15.
```

The user has miscounted and has input mults(5,0) twice. This is almost certainly an error. The parser now catches this error and issues something like the following warning message:

```
*** WARNING in line 261:
   mults(5,0) = 11. 13. 22. 15.
in file: test.in
A duplicate value has been specified for: mults(5,0) = 11.
This array element was first specified in line 260
   mults(0,0) = 0. 0. 1. 5. 6. 9.
in file: test.in
This warning can be turned into a fatal error with the command
   duplicate_array_values = fatal
Duplicate array value checking can be turned off totally with
   duplicate_array_values = none
This is not recommended since you will lose the opportunity
to check for legimate errors in your input.
```

The corrected input would presumably be (this is not part of the output message):

```
mults(1,1) = 0. 0. 1. 5. 6. 9.
mults(7,1) = 11. 13. 22. 15.
```

If, however, the user has the following input:

```
mults(1,1) = 0. 0. 1. 5. 6. 9.
mults(1,1) = 5. 5. 8. 9. 10. 19.
```

the parser will also flag this as a warning even though the user probably did this intentionally. It is recommended that the user comment out the first line so the parser can work as intended and catch input bugs as in the first example. Blocks of lines can be commented out with the /* ... */ multi-line comment characters.

The error severity for duplicate array values error checking can be changed by using the following commands:

```
duplicate_array_values = fatal   ! Duplicate array values are fatal
duplicate_array_values = warning ! Duplicate array values are warnings (default)
duplicate_array_values = none    ! Duplicate array values checking is not done.
```

Users are encouraged to examine the warnings carefully and make sure there are no input errors.

3.15 Stopping Processing

Normally parsing the user input file ends when the end of file is encountered. The user can, however, end the processing early in two ways using the following commands:

```
stop - end processing, continue calculation
fatal_error - issue error message, end calculation
```

The stop statement is allowed in user input files. When the stop statement is encountered, the compilation phase of handling the user input file stops immediately and *PowerParser* proceeds to setting code variables based the results of the compilation phase. Any commands appearing after the stop statement will be ignored, for example:

```
some_command = 5.0
stop
some_command = 6.0
```

The first line is processed normally and then the stop command stops the compilation phase. The third line is not processed. *PowerParser* will set the variable associated with some_command to 5.0.

The fatal_error command allows the user to print a message to the screen and end the calculation immediately. This is usually in response to a user detected error in the input file. The fatal_error command is followed (on the same line) by the error message to be echoed to the screen. Consider for example the following:

```
$rarg = ($bq**2 - 4*$aq*$cq)
if ($rarg .lt. 0.) then
   fatal_error sub quad_eq, sqrt argument < 0.
endif
$num = (-$bq + sqrt($rarg))
```

The $rarg variable is to be the argument to the sqrt function. The user would like to make sure that the sqrt argument is greater than or equal to zero, otherwise issue an informative message and end the calculation. The sqrt function itself would return a NaN (not a number) and the parser would end the calculation with some less than clear message about a NaN. But it is better to check for it in the input file since the error message is much more informative and will make it easier for the user to fix.

At this time, the error message for this example is:

```
*** User has issued a fatal_error command in line 776:
            fatal_error sub quad_eq, sqrt argument < 0.
in file: test.in

The user supplied fatal_error message is:
   sub quad_eq, sqrt argument < 0.
```

3.16 Debugging User Input

PowerParser has many features that help make superior input files, but that also means that users may occasionally have trouble writing their input files correctly. To help with this several debugging commands are provided which the user can use as an aid in developing input files.

Sometimes it is useful to know what variables have been defined and what their values are. Use the parser_list_variables command to list the defined variables and their values. This includes both pre-defined and user

defined variables. This command can be placed anywhere in the input file and can be used in multiple places in the file. Following is an example of output from the use of this command.

```
********** Debugging: list variable names and values
     Variable name          Value              Description
   ----------------- ----------------- ----------------------------
              $G           6.67428e-8      Grav con (cm**3/(g s**2))
             $Na        6.02214129e23     Avogadro's constant (/mol)
              $a     137.201704754335           a (erg/(cm**3 eV**4)
            $a23                  1.0
       $log1d(1)                 true      1d logical variable array
       $log1d(2)                false      1d logical variable array
              $r                  1.2
           $rho1                 1.34
          $sigma          5.670400e-5   Stefan B (erg/(s cm**2 K**4)
       $sigma_ev     1028300907752.31  Stefan B (erg/(s cm**2 eV**4)
            $sp2                 true
     $var2d(1,1)                  11.
     $var2d(2,1)                  21.
     $var2d(3,1)                  31.
********** Debugging END: list variable names and values
```

A single variable can be listed by following the parser_list_variables command with the name of the variable. This is useful for reducing the amount of output. For example:

```
parser_list_variables $var1
```

will list the value and description for the variable $var1.

At the end of processing the input file, the current list of variables (like $pi) is printed to the output file.

Sometimes it is useful to know what intrinsic functions are available. Use the parser_list_functions command to list the available intrinsic functions. This command can be placed anywhere in the input file and can be used in multiple places in the file. Following is an example of output from the use of this command (to save space some of the output has been removed):

```
********** Debugging: list function names
   Function name    nargs                Description
   -------------    -----    ---------------------------------------
        acos         1       arccosine, returns radians, arg -1 to 1
        asin         1         arcsine, returns radians, arg -1 to 1
        atan         1                  arctangent, returns radians
        ...
        tanh         1                            hyperbolic tangent
********** Debugging END: list function names
```

When *PowerParser* starts, the list of available intrinsic functions (like sin, cos, ...) is printed to the output file.

After the parser has "compiled" the user input file, it has created a final buffer of commands that the calling program actually uses to get input. It is sometimes useful to see this final buffer to debug the user input file. Use the "parser_print_fbuffer" command to print the final buffer after the user input file has been "compiled".

3.17 Restart Blocks

PowerParser has a feature called restart blocks which aid the user in doing calculations with multiple restarts. A restart block is a set of commands in the user input file delineated by a start command and an end command. A user specified condition, such as "time .ge. 50" is associated with the restart block (specified in the restart block start command). The restart block is initially ignored by the code. As the code runs, when the condition is met, *PowerParser* marks the

restart block as active, writes a restart dump, and gracefully ends the calculation. When the calculation is restarted all the commands in the restart block are parsed and used as normal commands. Multiple restart blocks are allowed and a variety of conditions are possible.

Note that some of this functionality is dependent on integration into the restart dump functionality of the simulation code. It may also work a little differently if the integration is not as assumed here.

Following is an example of a restart block

```
sizemat(1) = 0.1
restart_block air_zoning1 (time .gt. 10) then
   sizemat(1) = 0.0125
end_restart_block
```

In this example, sizemat for material 2 (air for example) starts at 0.1 cm. The restart_block command specifies a unique name for the block, "air_zoning1" in this example, and it specifies the condition, "time .gt. 10". When the simulation time gets greater than 10 seconds, the "air_zoning1 restart" block will be marked as active, a restart dump will be written, and the code will gracefully end. The calculation will subsequently be restarted using the run scripts/batch system, or restarted manually, or perhaps restarted using any other set of scripts. When restarting, effectively the "restart_block" and "end_restart_block" commands for the "air_zoning1" restart block are ignored and the code "sees" the following commands:

```
sizemat(1) = 0.1
sizemat(1) = 0.0125
```

Thus sizemat for material 2 is set to 0.0125 cm.

Restart blocks trigger anytime the condition changes. This is irrelevant in the above example, since once time is greater than 10 and triggers a restart, time will never fall to less than 10 and thus the above example will not trigger again. Consider, however, the following example:

```
shortmodcyc = 5
restart_block rb3 (ncycle .gt. 50 .and. ncycle .lt. 70) then
   shortmodcyc = 1
end_restart_block
```

The condition starts out as false and shortmodcyc is 5. When the simulation reaches cycle 51 the condition changes to true, and thus triggers a restart and shortmodcyc changes to 1. When cycle 70 is reached the condition changes from true to false and another restart is triggered. Since the condition is now false, the restart block is marked as inactive and the "shortmodcyc=1" command is not processed, thus shortmodcyc goes back to 5.

Any number of commands are allowed between the "restart_block" and "end_restart_block" commands. Any number of restart blocks are allowed, but each restart block must have a unique name.

When restarting, restart blocks can be added and removed from the user input file.

A single line version of the "restart_block" and "end_restart_block" commands are available, for the above example, it would be:

```
restart_block air_zoning1 (time .gt. 10) sizemat(2) = 0.0125
```

The following example shows the use of multiple restart blocks using the cycle number as the trigger:

```
restart_block my_name1 (ncycle .eq. 10) sizemat(5) = 0.0500
restart_block my_name2 (ncycle .eq. 20) sizemat(5) = 0.0250
restart_block my_name3 (ncycle .eq. 30) sizemat(5) = 0.0125
```

The program will restart at cycles 10, 20, and 30, resetting sizemat for material 5 at each restart. The restart block information, particularly the active flag, is stored on the restart dump and is not known until the restart dump is read. Thus the commands that are processed before the dump is read must not appear in restart blocks.

3.17.1 Restart Block Conditions

A simple condition consists of three words and is the following form:

```
variable relation value
```

for example, "time .gt. 1.35e-2". More generally, a condition consists of multiple subconditions linked together by logical operators, i.e.

```
variable relation value      logical      variable relation value      logical     &
   variable relation value      logical ...
```

Any number of subconditions are allowed. Three logical operators are allowed, they are:

```
.and., .or., .andthen.
```

The variable is some code variable that is allowed in the "restart_block" command. For example, time and ncycle. The relation is the relation between the variable and the value. Valid relations are:

```
.eq., .ne., .gt., .ge., .lt., .le.,
.hgeq., .hgne., .hggt., .hgge., .hglt., .hgle.
```

The value is the number to be tested against the variable. At this time, the condition value cannot be a parser variable or math expression. It can only be a number (or a logical or a string, see below).

The above examples all had floating point or integer conditions. A logical condition works the same way:

```
restart_block some_name (wttf_c01 .eq. TruE) then
   wt_cmd01 = true
   wt_cmd02 = 5.0
end_restart_block
```

Whereas "time" and "ncycle" are valid variables, the "wttf_c01" logical variable is not a valid code variable. At this time we have not implemented any logical or character variables, but perhaps will in the future.

For logical conditions, the only relations allowed are .eq. and .ne. and the only values allowed are true and false (or .true., .false., with any case).

Character conditions are allowed as in the following example:

```
restart_block my_name_3 (wttf_c02 .eq. "The Force") then
   wt_cmd03 = true
   wt_cmd04 = 6
end_restart_block
```

hg type relations. The hg in the hg type relations, such as .hggt., stands for "has gotten". For a subcondition that has a hg relation, for example .hggt., this means that when the code variable has gotten greater than the specified value, then that subcondition is marked as having been satisfied. It will remain marked as satisfied for the duration of the calculation.

For this to work, the subconditions have to be linked together with .andthen., not .and.

For example, suppose the user wants to restart and set the sizemat of material 6 to 0.01 when max_density drops below 1, but does not want to start checking for that until max_pressure has reached at least 50. This is accomplished with the following "restart_block" command:

```
restart_block hg_example (max_pressure .hgge. 50 .andthen. &
                          max_density .lt. 1) then
   sizemat(6) = 0.01
end_restart_block
```

The condition in this example consists of two subconditions:

```
max_pressure .hgge. 50
max_density .lt. 1
```

connected by the logical .andthen. operator. The first subcondition starts as being marked as not satisfied. Assuming the pressure starts at 0 and ramps up, eventually the max_pressure reaches 50 and then the first subcondition is marked as satisfied. During this ramp-up period, the max_density is checked but it does not matter since the first subcondition has not been satisfied.

Once the first subcondition, "max_pressure .hgge. 50" is marked as being satisfied, it will remain marked as satisfied even if max_pressure subsequently drops below 50.

Once the first subcondition is satisfied, then the result of the overall condition will be determined by the second subcondition "max_density .lt. 1". When max_density drops below 1, then the calculation will end and upon restart the sizemat of material 6 will be set to 0.01.

Note that "max_pressure" and "max_density" must be valid restart block condition variables in the simulation code for this to work.

3.17.2 Changing Restart Blocks

Restart blocks can be changed during a calculation, i.e., changed when a restart is done, however with certain constraints. How restart blocks can and cannot be changed is detailed as follows.

Deleting Restart Blocks Restart blocks can be removed from the input file at any time. If the restart block has already triggered, then the contents of the restart block will not be processed since it is gone from the input file. When the code discovers that a restart block exists on the restart dump, but is not in the input file, then it simply ignores the restart block on the dump.

Adding Restart Blocks Restart blocks can be added to the input file at any time.

Renaming Restart Blocks Restart blocks can be renamed at any time, subject to the new name being unique. This is equivalent to removing a restart block and adding in a new one.

Changing Restart Block Content The restart block content can be changed at any time. If the restart block has already triggered, the changed content will be used.

Changing Restart Block Conditions If the restart block has already triggered then the condition cannot be changed. If the restart block has not triggered then the condition can be changed, however the number of sub-conditions in the condition cannot be changed.

3.18 When...Then

PowerParser also has a feature called the when..then command. The when...then command is put in the user input file as shown:

```
when some condition is satisfied then
   process some commands
endwhen
```

The when...then condition is checked every cycle as the calculation runs, when the condition is satisfied, the associated commands are executed. The when...then command is then tagged as inactive and is not checked again, except when doing a restart where it is checked again. See the discussion of when...then restarts below.

A related command is the whenever command:

```
whenever some condition is satisfied then
   process some commands
endwhen
```

The whenever command is the same as the when command, except that when the whenever condition is satisfied and the commands are executed, it is not tagged as inactive as opposed to the when command which is tagged as inactive. Thus the whenever command is checked every cycle and the commands will be executed whenever the condition is satisfied. The commands for the when command will only be executed the first time the condition is satisfied.

See below for allowable conditions and commands (applies to both when and whenever).

Any number of when...then (and whenever) commands are allowed in the input file. Each one is treated separately.

For example, suppose the user wants to change the shortmodcyc frequency to 5 when the simulation time reaches 5.0. This can be done with the following:

```
when (time .gt. 5.0) then
   shortmodcyc = 5
endwhen
```

This condition consists of three words and is the following form:

```
variable relation value
```

More generally, a condition consists of multiple subconditions linked together by logical operators, i.e.

```
variable relation value     logical     variable relation value &
   logical    variable relation value    logical ...
```

Any number of subconditions are allowed. Three logical operators are allowed. They are:

```
.and., .or., .andthen.
```

The variable is some code variable that is allowed in the when...then command. For example, time and ncycle. The relation is the comparison operator between the variable and the value. Valid relations are:

```
.eq., .ne., .gt., .ge., .lt., .le.,
.hgeq., .hgne., .hggt., .hgge., .hglt., .hgle.
```

The meaning of the hg type relations, such as .hggt., were discussed earlier in the Restart Blocks Section.

The value is the number to be tested against the variable. At this time, the condition value cannot be a parser variable or math expression. It can only be a number (or a logical or a string, see below).

When there is only one command to be executed, then the when...then command can be put on one line as in the following example:

```
when (time .gt. 5.0) shortmodcyc = 5
when (time .gt. 10.0) shortmodcyc = 1
when (time .gt. 11.0) shortmodcyc = 5
when (time .gt. 20.0) shortmodcyc = 50
```

The above examples all had floating point conditions. An integer condition works the same way:

```
when (ncycle .gt. 10) modcyc = 5
```

Logical conditions also work:

```
when (wttf_c01 .eq. TruE) then
   wt_cmd01 = true
   wt_cmd02 = 5.0
endwhen
```

For logical conditions, the only relations allowed are .eq. and .ne. and the only values allowed are true and false (or .true., .false., with any case).

Character conditions are allowed as in the following example:

```
when (wttf_c02 .eq. "The Force") then
   wt_cmd03 = true
   wt_cmd04 = 6
endwhen
```

hg type relations. The hg type relations described in the Restart Blocks can also be used in when...then commands. For example, suppose the user wants to set the shortmodcyc to 10 when max_density drops below 1, but does not want to start checking for that until max_pressure has reached at least 50. This is accomplished with the following whenthen:

```
when (max_pressure .hgge. 50 .andthen. max_density .lt. 1) then
   shortmodcyc = 10
endwhen
```

The condition in this example consists of two subconditions:

```
max_pressure .hgge. 50
max_density .lt. 1
```

connected by the logical .andthen. operator. The first subcondition starts as being marked as not satisfied. Assuming the pressure starts at 0 and ramps up, eventually the max_pressure reaches 50 and then the first subcondition is marked as satisfied. During this ramp-up period, the max_density is checked but it does not matter since the first subcondition has not been satisfied.

Once the first subcondition, "max_pressure .hgge. 50" is marked as being satisfied, it will remain marked as satisfied even if max_pressure subsequently drops below 50.

Once the first subcondition is satisfied, then the result of the overall condition will be determined by the second subcondition "max_density .lt. 1". When max_density drops below 1, then the commands will be executed and shortmodcyc will be set to 10.

When...then restarts. Several aspects of the when...then commands are stored on the restart dumps, including the when...then active flags. When restarting the when...then commands will all be processed again in the order that they were triggered. When restarting, the user can remove and/or add when...then commands.

Allowable when...then conditions and commands. The allowed condition variables and commands are dependent on the integration of *PowerParser* into the application code.

CHAPTER

FOUR

EXAMPLES

4.1 Example 1

This example will go through writing a quadratic equation subroutine in the input file. One reason for doing this is to use more complicated examples to help find parser bugs before the users find them. The second reason is to provide a step by step example of writing a subroutine including the problems encountered.

The quadratic equation is:

$$ax^2 + bx + c = 0$$

and its solution is:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

We start the quadratic subroutine with the following line:

```
subroutine quad_eq($a, $b, $c, $root1, $root2)
```

Next, it is good practice to initialize the output variables.

```
$root1 = 0.
$root2 = 0.
```

Then we define the argument to the sqrt function and make sure it is greater than or equal to 0. It is good practice to make checks like this in the user input file so as to get more informative error messages and make it easier to fix the input file.

```
$rarg = ($b**2 - 4*$a*$c)
if ($rarg .lt. 0.) then
   fatal_error sub quad_eq, sqrt argument < 0.
endif
```

Now define the numerator and denominator for the roots and make sure that the denominator is not 0.

```
$num = (-$b + sqrt($rarg))
$dem = (2 * $a)
if ($dem .eq. 0.) then
   fatal_error sub quad_eq, denominator (2a) = 0.
endif
```

Finally calculate the roots.

```
   $root1 = $num / $dem
   $num = (-$b - sqrt($rarg))
   $root2 = $num / $dem
end subroutine
```

Notice that a mistake has been made in calculating the roots. The parentheses around the math expressions are missing. This is one of the more common errors in developing input files. Another common error is missing dollar signs in front of variable names. The corrected input is:

```
   $root1 = ($num / $dem)
   $num = (-$b - sqrt($rarg))
   $root2 = ($num / $dem)
end subroutine
```

The final subroutine is given here:

```
!==============================================================================
! Subroutine to solve the quadratic equation.
!==============================================================================
subroutine quad_eq($a, $b, $c, $root1, $root2)
   $root1 = 0.
   $root2 = 0.
   $rarg = ($b**2 - 4*$a*$c)
   if ($rarg .lt. 0.) then
      fatal_error sub quad_eq, sqrt argument < 0.
   endif
   $num = (-$b + sqrt($rarg))
   $dem = (2 * $a)
   if ($dem .eq. 0.) then
      fatal_error sub quad_eq, denominator (2a) = 0.
   endif
   $root1 = ($num / $dem)
   $num = (-$b - sqrt($rarg))
   $root2 = ($num / $dem)
end subroutine
```

The first attempt at calling the quadratic subroutine was:

```
call quad_eq(2, 4, -30, $x1, $x2)
```

This had 2 problems. The first is that $x1 and $x2 have not been defined. It is natural to want to use them in this way since we know that the subroutine is going to define them. But the current parser will not allow that. This perhaps will be fixed sometime. The corrected input is:

$x1 = 0 $x2 = 0 call quad_eq(2, 4, -30, $x1, $x2)

The second problem is that there was a bug in the parser that the unary minus sign in front of the number 30 is not being applied correctly. This has been fixed. Unfortunately problems like this might crop up as the parser is being debugged. Users are encouraged to report the bug and to attempt a workaround. In this case, treating the -30 as a math expression and enclosing it in parentheses worked, i.e.

```
$x1 = 0
$x2 = 0
call quad_eq(2, 4, (-30), $x1, $x2)
```

It would have also worked to define a new variable equal to -30 and use the variable in the subroutine call.

For this example, the roots, $x1 and $x2, end up being 3 and -5.

It is good practice to verify that the fatal_error commands work as expected. The division by zero can be checked by calling the subroutine with $a = 0

```
call quad_eq(0, 4, -30, $x1, $x2)
```

The sqrt of a negative number error can be checked by changing -30 to +30.

```
call quad_eq(2, 4, 30, $x1, $x2)
```

CHAPTER

FIVE

DEVELOPER'S GUIDE

The *PowerParser* package is designed to handle the input file parsing needs for simulation codes. The interface is in C++ with a parallel and serial version of the library. Operations include commands such as simple key,value input:

```
dx = 1.0
```

as well as array operations, mathematical operations, and restart support.

Each input line is called a command in the *PowerParser* nomenclature and the left hand side of the expression is a key and the right hand side is the value.

5.1 Parser Initialization

The parser can be initialized with one of the two following commands:

```
#include "PowerParser.hh"
using namespace PP;

int main(int argc, char *argv[])
{
   .....

   PowerParser *parse = new PowerParser();
      or
   PowerParser *parse = new PowerParser("file.in");
```

This operation will initialize the parsing object and either initialize MPI or use an already initialized MPI context. The processor communication layer is automatically handled, but a user can retrieve some MPI settings such as number of processors or rank with:

```
int mype = parse->comm->getProcRank();
int npes = parse->comm->getNumProcs();
```

Of course, a developer can get these settings by querying MPI directly.

Initiating the parsing of an input file can be done with:

```
parse->parse_file("parsetest.in");

parse->compile_buffer();
```

These calls perform the following operations

- Reads the input file into a deque of lines
- Breaks the lines up into words

- Compiles the input buffer

The user input buffer contains execution line arguments, the input files, and any files included by them. Compiling the user input buffer handles loops, subroutines, if statements, variables, etc. The end result is a final buffer that can be queried by *get* commands.

There are several functions to echo out the input, the final buffer after compiling and intermediate variables and functions. Some of the commands and their purpose are:

```
parse->echo_input_start();
```

TODO: We need a lot more detail on the outputs.

TODO: Need description of debugging checks

5.2 Using the Parser

5.2.1 Scalar input

The retrieval of simple key-value pairs is straight-forward, such as the following:

```
int intvalue = -1;
parse->get_int("int_input", &intvalue);
```

This code looks for a simple key value pair in the parse buffer such as the following:

```
int_input = 8
```

The get_int call returns the intvalue variable with it set to the value 8. There are a couple of things to note. You should set the value of the variable before the get call, because it is set only if there is a key of "int_input" in the input file. It would end up uninitialized if the key is not present and could cause indeterminate behavior. Also, the key string is case sensitive and so "int_input" and "Int_Input" are different keys.

The complete list of scalar get functions are:

```
//Character array versions
   parse->get_bool    (const char *cname, bool    *cvalue)
   parse->get_bool_int(const char *cname, int     *cvalue)
   parse->get_int     (const char *cname, int     *cvalue)
   parse->get_int64_t (const char *cname, int64_t *cvalue)
   parse->get_real    (const char *cname, double  *cvalue)
   parse->get_char    (const char *cname, double  *cvalue)

// String versions
   parse->get_bool    (string &cname,     bool    *cvalue)
   parse->get_bool_int(string &cname,     int     *cvalue)
   parse->get_int     (string &cname,     int     *cvalue)
   parse->get_int64_t (string &cname,     int64_t *cvalue)
   parse->get_char    (string &cname,     double  *cvalue)
```

These functions also take two possible optional arguments:

```
const vector<int> &size -- set to {i}, {i,j}, {i,j,k}, etc
                           for arrays and multi-dimensional arrays
                           default = NULL, means scalar input

bool skip = false -- has to do with skipping assignment to help
                     handle restarts
```

The get_char functions also have an optional variable called single_char that can be true or false.

5.2.2 Vector input

For an array input, the code would like this:

```
vector<int> size = {6};
double doublearray[6] = {-1.0, -1.0, -1.0, -1.0, -1.0, -1.0};
parse->get_real("array1d", doublearray, size);
```

or for a 2D array:

```
vector<int> size = {3,2};
double **doublearray2d = (double **)genmatrix(size[1], size[0], sizeof(double));
for (int j = 0; j < size[1]; j++){
   for (int i = 0; i < size[0]; i++){
      doublearray2d[j][i] == -1.0;
   }
}
parse->get_real("array2d", &doublearray2d[0][0], size);
```

The genmatrix routine is a special allocator from the genmalloc package that allocates a contiguous block of data for multi-dimensional arrays and then assigns the pointers to the correct places in the block of data to work correctly with multiple indices.

Not all the time is the size of the input know before-hand. For the 1D array above, it often is better to query the size before allocating and reading the array:

```
vector<int> size;
parse->get_size("array1d", size)
double *doublearray = (double *)malloc(size * sizeof(double));
for (int i = 0; i < size[0]; i++){
   doublearray[i] = -1.0;
}
parse->get_real("array1d", doublearray, size);
```

Up to 4D arrays are currently supported in *PowerParser*.

5.2.3 Key in input

Sometimes it is useful to query the parser to find out if a particular key appears anywhere in the user input. This is done with the cmd_in_input function. For example, suppose we want to know if the "special_variable" key is in the user input. We would make the following call:

```
string cname("special_variable");
bool in_input = false;
bool in_whenthen = false;
parse->cmd_in_input(cname, in_input, in_whenthen);
```

The logical variable "in_input" will be true if "special_variable" is found anywhere in the main input, not including the when...then blocks. The logical variable "in_whenthen" will be true if "special_variable" is fond in any of the when...then blocks.

The search for the key is done on the final buffer and this makes it certain that it will exist if a get command for the key is done.

5.3 Parser Checks

After all the input routines have been called, the developer should check that all the user commands in the user input file were processed. If any command, or part of a command, was not processed, then a fatal error is produced. This check is done with the following call:

```
logical :: good
parse->check_processed(good)
```

If a fatal error is generated, the code will end.

The assumption is being made that if a user command or part of a command is not processed then it is most likely a user error, the user mistyped something, the user does not understand the input, etc. In any case, the user is expected to fix the input file before it can be successfully run.

This puts a burden on the developer. When the user turns on an option, it is expected that the commands associated with that option can remain in the input file even though the option is off and the commands are not really needed. Thus the developer needs to make certain that all get commands associated with the option are processed even when the option is turned off by the user.

Consider how the example of the size query should be changed if the package is turned off and yet we still want to process the get call. We can use the optional skip argument to process the input, but not set the variable, as follows:

```
vector<int> size;
parse->get_size("array1d", size)

if (package_is_off) {
   parse->get_real("array1d", doublearray, size, true);
} else {
   double *doublearray = (double *)malloc(size * sizeof(double));
   for (int i = 0; i < size[0]; i++){
      doublearray[i] = -1.0;
   }
   parse->get_real("array1d", doublearray, size, false);
}
```

A better way to handle this is to use the cmd_set_processed routine to set the processed flag for the command and its arguments to true. For example:

```
vector<int> size;
parse->get_size("array1d", size)

if (package_is_off) {
   parse->cmd_set_processed("array1d", true);
} else {
   double *doublearray = (double *)malloc(size * sizeof(double));
   for (int i = 0; i < size[0]; i++){
      doublearray[i] = -1.0;
   }
   parse->get_real("array1d", doublearray, size, false);
}
```

The cmd_set_processed call takes two arguments; the first is the name of the command and the second is the setting for the processed flag, either true or false.

5.4 Parser Error Handling

PowerParser does an exceptional job at handling and reporting errors. When an error occurs, the parser reports the line number in the user input where the error occurred, echos the line, reports what file the line is found in, and gives a detailed description of the error. At this time, all parser errors are fatal errors, and *PowerParser* will abort the run.

Most of the time, the errors are accumulated and are reported at the end of the parsing process. This is good in that the user can correct more than one error at a time. It is bad in that the errors can "snowball", in that one error can potentially generate many spurious errors. In such cases the user only needs to fix the first error to get rid of all the errors.

The file name is reported for an error because of the include capability of the parser, the user can include external input files at any place in the main input file. The line number is always the line number for the included file, or the main input file. This makes it easy for the user to find the file and line where the error occurred.

When there are continuation lines in the input file, *PowerParser* combines those into one line before processing. But the error reporting is always done for the continuation line and not for the combined line. Consider the following line:

```
$sp2 = true
mult_logical_array(1) = 3*false &
                        2*$sp2, &
                        .truezz.
```

There is an error in the last continuation line, true is misspelled. The parser reports the following error:

```
Fatal errors have been encountered while parsing the user input file.
Note that often fixing the first error will also fix the other errors.

*** FATAL ERROR in line 175:
                              .truezz.
in file: test.in
Values on this line should be true or false (or .true. or .false.)
   (any case is fine, for example true, True, TrUe are all ok)
Instead found value: .truezz.
```

5.5 When...then

The first step in implementing the when...then code is to get the number of when...then commands, wtnum, in the example below:

```
int wtnum = 0;
parse->whenthen_num_cpp(wtnum);
```

For every cycle in the simulation code, the following steps need to be done. First setup two arrays called code_varnames and code_values. For example:

```
for (int i=1; i<=wtnum; i++) {
   parse->whenthen_check_cpp(i, code_varnames, code_values, wt_check);
   if (wt_check) {
      parse->cpp("shortmodcyc", shortmodcyc, true);
   }
}
```

The code_varnames and code_values are strings. The whenthen_check routine checks to see if the condition has been satisfied. The first argument to this routine is the when...then sequence number (starting from 1) as determined by the order of the when...then commands in the user input file. The second argument is the name of the simulation code internal variable that is to be checked. The third argument is the value of the condition variable that is the value of the

internal code variable. The fourth argument is the true or false result output from this routine, called wt_check in this example. If wt_check is returned as true, then the condition was satisfied, otherwise it was not satisfied.

Note that the when...then class contains a processed flag, defaulted to false, which is set to true the first time the condition is satisfied. Thus, after the condition has been satisfied, subsequent calls to whenthen_check for this particular when...then command will always return false.

The related whenever command does not set the processed flag.

When the condition is first satisfied, wt_check is true. Then the various calls can be made. In the above example, the internal simulation variable shortmodcyc can be changed by the user.

After cycling through all the when...then commands, the following call must be made:

```
parse->whenthen_reset();
```

This resets the pointer to the final commands buffer back to the main buffer.

CHAPTER

SIX

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*