QuARTIC® v1.1.16 User Guide
ISL-ESD-TR-23-14

QuARTIC®
Qualified Automated Reporting Tool for Inputs and Calculation notebooks

Lance Larsen

Information Systems Laboratories, Inc.
Idaho Falls, ID

September, 2023

Table of Contents

1 Introduction 1

2 Using QuARTIC® 4

2.1 Installation . . . 4
2.2 Running QuARTIC® . . . 4
2.2.1 QuARTIC® Messages . . . 6
2.3 The Parameters Database . . . 6
2.3.1 Specifying Arrays . . . 9
2.3.2 Param Naming Conventions . . . 12
2.3.3 Adding Parameter Database Commands . . . 12
2.4 Template Files . . . 14
2.4.1 Latex . . . 14
2.4.2 Template File Syntax . . . 14
2.4.2.1 Code Blocks . . . 15
2.4.2.1.1 Code Block Modifiers . . . 15
2.4.2.2 Equation Blocks . . . 17
2.4.2.2.1 Equation Block Functions . . . 18
2.4.2.2.2 Equation Blocks Create 'Param' Objects . . . 20
2.4.2.2.3 Equation Block Modifiers . . . 21
2.4.2.2.4 Equation Style . . . 23
2.4.2.2.5 Inserting latex between equations . . . 25
2.4.2.2.6 Explicit Parenthesis . . . 25
2.4.2.2.7 Handling Long Equations . . . 26
2.4.2.3 Comment Blocks . . . 27
2.4.3 Including External Template Files . . . 27
2.4.4 Dynamic Template Functions . . . 28
2.4.5 Including Hooks . . . 29
2.4.6 Lua Functions Available in Template File . . . 31
2.4.7 Event Listener Functions . . . 36
2.4.7.1 The `onSaveParam` Event . . . 37
2.4.7.2 The `onValue` Event . . . 37
2.4.8 User Defined Modules . . . 37
2.4.9 Autoloaded modules . . . 38
2.4.10 Lua Objects . . . 38
2.4.10.1 _acronyms Object . . . 38
2.4.10.2 _event Object . . . 39
2.4.10.3 Param Object . . . 41

2.4.11 QuARTIC Modules 42
2.4.11.1 RegisterUnits Module 43
2.4.11.2 InputTable Module 43
2.4.11.3 RELAP5 Parser Functions and RELAP5 Database Objects 44
2.4.11.4 RELAP5 Documentation Modules 48
2.4.11.5 ParamTable Module 56

3 Debugging QuARTIC Template Errors 62
3.1 Example of Fixing Error Reported by Lua Script 63
3.2 Using the Builtin Lua Debugger 64
3.3 Example of Fixing Error Reported by Latex 65

1 Introduction

QuARTIC® is a tool for including live calculations in an engineering calculation document. It was specifically designed for analytical model development to maintain consistency between design information, the model calculation notebook, which is used to calculate the various parameters used in a simulation model, and the model itself. Modules are available for integration with analysis tools like TRACE and RELAP5.

Maintaining consistency between the design information, the calculation notebook, and the model, in order to maintain high quality standards, is typically a time consuming and error prone process. QuARTIC® was designed to make that process less error prone and labor intensive so that the burden of quality assurance is significantly reduced allowing projects to progress more quickly with higher confidence in the quality of the models that are developed.

A typical method of model development and documentation includes the following process, using a TRACE model as an example. First the information needed to build the model, such as plant geometry and materials, reactor kinetics parameters, safety system control logic information, etc, is gathered. Next, the information that was gathered in the first step is used to calculate required TRACE model inputs, such as volumes, flow areas, cell lengths, etc. The calculated data is then used to populate the TRACE model components. Documentation of the model often follows as a separate step. This typical approach to TRACE model development is depcited in Figure 1.1. This includes three primary deliverables:

1. The calculations performed to generate the model perhaps performed in a tool like excel.
2. Documentation of the calculations used to generate the model.
3. The model that is used to perform simulations.

The circular boundaries in Figure 1.1 represent interfaces between the deliverables in the the typical process where manual effort is required in order to keep the deliverables consistent. This approach introduces significant potential for the following types of errors:

- Transcription errors may occur, where values are not transferred correctly between the different deliverables.
- Calculation dependency errors can occur where a value is update in the calculations, but values in dependent calculations are missed and not updated appropriately.
- Changes in the equations used in the calculation tool may not get transferred to the documentation.
- Failure to transfer changes that occur in one of the deliverable items to the other deliverables.

Because of the potential for errors, significant time must be spent in the review process *just to verify*

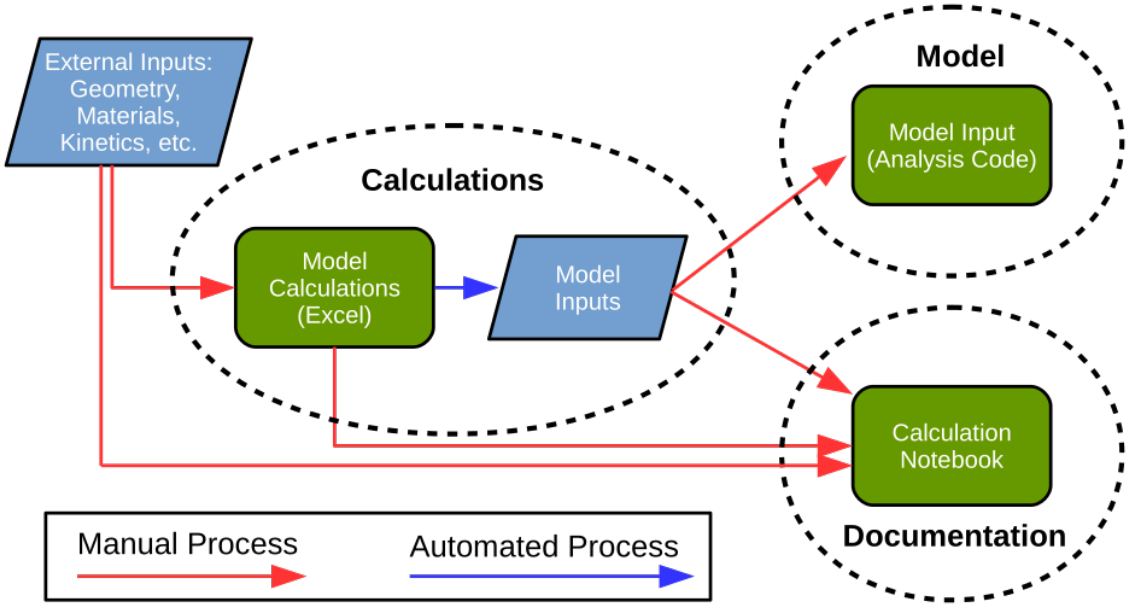

Figure 1.1: Typical Model Development Process

consistency. Keeping the deliverables consistent proves to be a difficult challenge, and review can be a time consuming (costly) process. Because of the burden of review, making small improvements to the model can require significant effort.

QuARTIC® improves this process significantly by:

1. Including inputs in a parameters database (spreadsheet document), that becomes a single point to references for all inputs needed to develop the model. Each parameter is given a unique variable so that the parameter can be used in symbolic calculations.

2. Allowing symbolic calculations, which are expanded using values from the parameters database, to be embedded directly in the documentation rather than in an external tool like excel.

3. Ouput values that are model parameters are identified and written to an CSV file which provides a summary of the key calculated values.

4. QuARTIC® modules are available for some tools that allow QuARTIC® to automatically insert calculated parameters into the model to simplify the process of maintaining consistency between the documentation and the model.

The improved process is depicted in Figure 1.2.

The process of gathering the information necessary to build the model is still a manual process, as well as the process of setting up the symbolic equations in the calculation notebook. But there is no longer the need for a separate tool for performing calculations and documenting the calculations, so the calculation notebook is consistent with the actual calculations by definition. In addition, for analysis tools with a supporting QuARTIC® module, the model parameters calculated in the

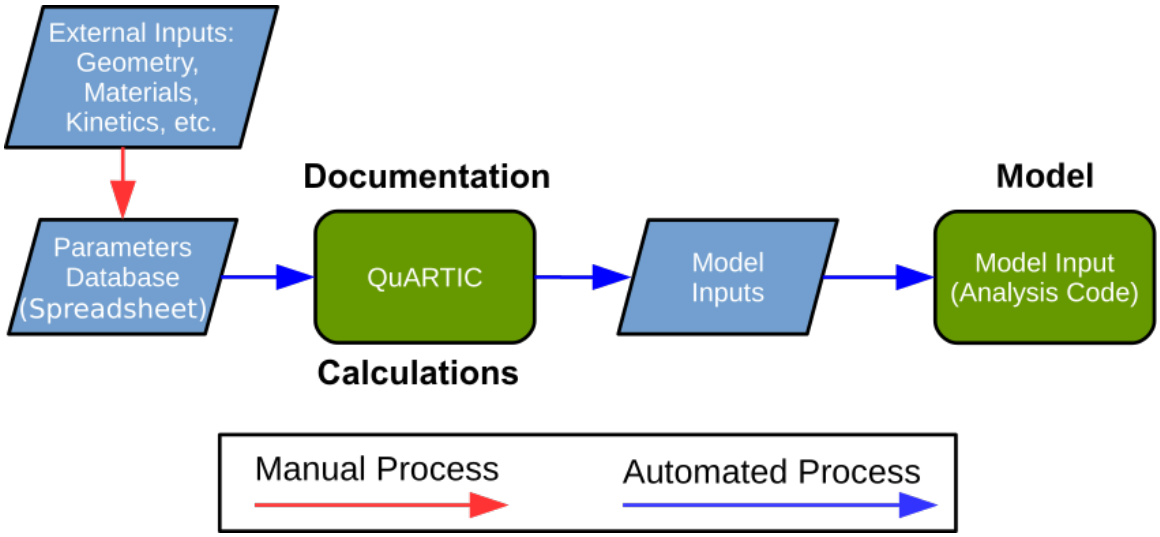

Figure 1.2: Improved TRACE Model Development Process

notebook can be configured to automatically populate the model. In order to achieve this goal, QuARTIC® is built on top of the following technologies:

1. Parameters are read from a set of CSV files which may be exported from a spreadsheet tool such as excel or libreoffice calc. Details about the format of the parameters database can be found in Section 2.3.

2. The calculation notebook is written in latex format with embedded scripting for calculations and other automation tasks. There are a few different programs available that process latex files and convert them to PDF. For QuARTIC®, the lualatex processor is used. Latex is a powerful text based technical documentation language that is commonly used for technical papers and books. Many resources for writing latex can be found on the internet. The latex file with embedded scripting is referred to as a template file in this documentation since the template file must be preprocessed by the QuARTIC® scripts to generate an actual latex file that can then be processed by lualatex to generate the final PDF document. Section 2.4 provides instructions for building latex template files.

3. Lua is used as an embedded scripting language inside the template files for generating dynamic content (such as calculations). The Lua commands that are available for use within a template are described in Section 2.4.6.

2 Using QuARTIC®

In order to use QuARTIC®, one or more QuARTIC® template files need to be created. In QuARTIC®, just as with latex, a document can be divided into multiple files which are combined in the final PDF document. The instructions for creating template files are provided in Section 2.4, with information about some of the QuARTIC® features described in Section 2.4.10.

In order to generate the final PDF document, two steps are required. First the template files must be processed by QuARTIC® in order to generate the final latex files. Then the latex files must then be processed by a latex processor to generate the final pdf.

It is recommended that both the template files and the final latex documents be maintained in a source control repository as part of the QA process. A source control repository is used to track changes to a document and see exactly what has been modified. Examining changes to the final latex documents between versions can be helpful for QA purposes to verify that all document changes are intended and are accurate.

2.1 Installation

In order to use QuARTIC®, the QuARTIC® executable and a latex processor must be installed. Lualatex is a tool that processes latex documents, and is recommended for processing the latex documents generated by QuARTIC®. QuARTIC® has been test incombination with the lualatex v 0.95 from the texlive 2016 distibution, but is expected to work with other versions of latex. Texlive is an opensource collection of tools for processing latex documents and can be downloaded from https://www.tug.org/texlive/. This site contains instructions for installation on windows, linux, and MacOS.

2.2 Running QuARTIC®

The QuARTIC® executable is used to convert QuARTIC® template files (latex with embedded calculations and scripting) into latex files. After QuARTIC® is run, the latex file that is generated must be processed by lated to generate a pdf document. The basic format for executing QuARTIC® is:

```
> QuARTIC.exe [options] <template_file>
```

The command line options listed below are available for the QuARTIC® script.

-a <command line args file> A file containing command line arguments to use.

-c <csv input dir> Specify the location of the csv parameters database files. All CSV files in the folder will be read in.

-init <CSV init file> This is a CSV file that contains any values needed before processing other CSV files. This can be used to set preferred units so that parameters with with units will automatically be converted to preferred units as they are loaded (default init.csv).

-loglvl <log level> Set the log level, which controls the type of information that is logged to the screen. Higher values result in more information being logged. The log levels are 1=ERROR, 2=WARNING, 3=INFO, 4=DEBUG.

-h Print the help message.

-H Show hidden equations.

-m Mark output parameters. This causes the parameters to be shown in **bold magenta** format when they are calculated.

-o <latex output directory> The location where the final latex files will be stored after processing the template files (default ../Latex/)

-p Print items loaded from the CSV input files. Note that the -c option must be specified in order for this option to function.

-t <template directory> Path to the latex template files (default ../tmp/)

-TI <TRACE input file folder> Path to the TRACE input file.

-Ti <TRACE input file name> The name of the TRACE input file to update using parameter calculated in the calculation notebook.

-TO <TRACE output folder> Pathe where updated TRACE input files will be written (default is the TRACE input folder specified via -TI)

-To <TRACE output file name> This is the name of the updated TRACE input file that is written to te TRACE output folder.

-Tp <TRACE output param file> This is the TRACE input file with variable names inserted in place of values. This is use to verify that values are inserted in the correct location. It is also used to check that values calculated in the calculation notebook are inserted as expected.

-X Flag to turn off compilation of the template files. This is used when manually editing the template files for debug purposes.

When running from the command line, it is inconvenient to pass in a long list of command line parameters. By default, QuARTIC® will look for a file named 'QuARTIC.ini' and load default parameters from this file if present in the folder where QuARTIC® is executed as long as no

command line parameters are passed to QuARTIC. The command line parameters file can be overriden by passing in a **'-a <args file>'** option which defines a different args file to read in and forces the file to be read even if command line parameters are used.

2.2.1 QuARTIC® Messages

As QuARTIC® runs, it prints various messages. Some messages indicate the current task being performed by QuARTIC®, such as loading parameters from a CSV file. Other messages may indicate warnings or errors. The '-loglvl' command line option, described above, is used to control the amount of detail that is printed to the console by QuARTIC®.

2.3 The Parameters Database

The parameters database is used to initialize data that will be used in the calculation notebook. The parameters database is a set of CSV files that are processed by QuARTIC® prior to compiling and running the template files. The CSV files are best maintained in a spreadsheet document and exported as CSV. Macros are available for excel and libreoffice that export spreadsheet tabs as CSV files.

QuARTIC® expects the parameter database to be organized into command tables. A command table starts with a command header row, followed by a table of commands of the type indicated by the header row. The command header row indicates the type of command in the first column. The columns that follow on the header row indicate the name of command fields. Command fields can be placed in any row desired and do not have to be specified in any particular order. Columns can be left blank if this is convenient for organization. The rows that follow a command header are instances of the command and generally do not include a value in the first column. The command header followed by the associated commands in referred to as a command table.

Comment lines can be added by putting a '#' character in the first column. Blank lines are ignored.

A single csv file can contain multiple command tables, with each command table starting with a command header row. QuARTIC can also read multiple csv files from the csv folders that are indicated on the command line.

Each command type supports a specific set of fields, some of which are necessary and some which are optional. Custom fields can also be added if useful. Table entries follow the header row. Each table entry will leave the first column blank, and will fill in values as appropriate for the fields identified in the header row. The following command types are supported:

Table 2.1: Parameter Database Commands

Command/Fields	Description
acronyms	A table of acronyms and definitions for acronyms used in the parameter descriptions
- acronym	The acronym or abbreviation.
- definition	Definition of the acronym or abbreviation
figure	Information about a figure that may be used in the document. If a figure command is included, then the function `addFigure(name, options)` will be available for adding the figure to the document, where the name is the name field value indicated below, and options can include 'figureFolder', 'caption', 'width', 'height', and 'iscapbelow' to override optional fields specified below.
- name	The name used to request the figure via `addFigure`. The label for referencing the figure is 'fig:<name>'.
- figname	The filename of the figure.
- caption	(Optional) A figure caption (using latex syntax).
- figureFolder	(Optional) The folder where the figure is located relative to the latex folder that is specified on the command line. A 'figureFolder' variable can be specified in the template, and will be applied to all `addFigure` commands that follow that do not have 'figureFolder' specified as a field.
- width	(Optional) A latex figure width specification (e.g., '5in').
- height	(Optional) A latex figure height specification (e.g., '4in').
- iscapbelow	(Optional) Indicate if the caption is below (or above) the figure. Default is `true`, but the value can be set to 'false' (or 'f' or 'n') to put the caption above the figure.
parameter	Define a parameter (or variable) to be used in the document.
- name	The parameter name. Some special conventions apply to the name. This will be discussed in Section 2.3.2.
- value	The value associated with the parameter. This can be a number or a string.
- units	(Optional) This column is used to specify the units for the parameter. Note that if a compatible unit is specified in the preferred units, the value is converted to preferred units for use in the calculation notebook. To override this, set the calcunits field.
- ref	(Optional) Short name of the reference the parameter comes from.
- loc	(Optional) Specific location of the parameter within the reference.
- desc	(Optional) Description of the parameter.

Continued on next page

Table 2.1: Parameter Database Commands (Continued)

Command/Fields	Description
- acronyms	(Optional) An '|' separated list of acronyms used in the parameter description. This is used for detecting which acronyms should be included in the autogenerated acronyms table.
- fmt	(Optional) Format of the numeric value. This can be used to control the number of significant digits for a variable. The format follows the Lua string convention for formatting numerical values.
- tex	(Optional) Optional latex representation of the variable (in math mode syntax). Note that QuARTIC® does create a default latex representation for variables. This is described in Section 2.3.2.
preferred	Specify preferred units for a specific unit type
- units	The preferred units. For example the preferred units for length might be 'm' (meters) or 'ft' (feet). The preferred unit can be overridden for a parameter by specifying the 'calcunits' field.
references	A list of references used by parameters in the parameter database. When a reference command is included the functions `cite(ref,before,after)` and `References()` are available to use in the template. The `cite` is passed the ref value below and marks the reference as being used. It adds a latex citation at the location where it is used. The before and after values specify the text that comes before and after the citation. The `References` function adds a bibliography section that includes all references that were marked as being used in the document. Typically, `References` will be called without parenthesis. This causes the function to not be evaluated until after the full document is processed. That way all references are tracked properly even if they occur after the references section is added.
- ref	A short name for the reference. Most parameters should include a reference, and the short 'ref' name is used for the reference.
- title	The title for the referenced item.
- doc_id	(Optional) A document id perhaps from an external reference system.
- doc_auth	(Optional) Author(s) of the document.
- rev	(Optional) The document revision information.
- doc_url	(Optional) Url to the reference source if applicable.
- doc_num	(Optional) Document number if applicable.
- doc_date	(Optional) The date the reference was published.
- desc	(Optional) A description of the document.

One and two dimensional arrays can also be included in the parameters database via the 'array' command. However, the format of arrays is not consistent with some of the conventions discussed

above. So array commands are discussed further in Section 2.3.1.

The following example shows how commands may be organized in a spreadsheet. This spreadsheet can then be exported as a CSV file.

Table 2.2: Parameter Database Commands

# Comments can be added as lines that start with a '#' symbol.						
acronyms		**acronym**				**definition**
		LC				QuARTIC
		PD				Pressure Dome
		XSEC				Crossover Section
parameter	**name**	**value**	**units**	**acronym**	**ref**	**desc**
	`Vol_PD`	220	`m^3`	LC|PD	DOC	LC volume of the PD
	`Len_PD`	220	m	PD	DOC	Length of the PD
	`Vol_XSEC`	10.3	`m^3`	XSEC	DOC	Volume of the XSEC
preferred	**unit**					
	`cm^3`					Preferred Volume (comment)
references	**ref**	**rev**				**title**
	DOC	1				Reference document title

2.3.1 Specifying Arrays

It is common to have a set of one dimensional arrays that are the same length that contain related data. The 'array' command can be used to construct a set of related one dimensional arrays. This can be organized in a spreadsheet as follows:

Table 2.3: One Dimensional Array Command

array	<array 1 name>	<array 2 name>	...
units	<array 1 units>	<array 2 units>	...

Table 2.3: One Dimensional Array Command

	<value 1>	<value 1>	...
	<value 2>	<value 2>	...
	<value 3>	<value 3>	...
	...	...	...

The array names must conform to the Param naming convention (Section 2.3.2). In some cases, it is useful to provide a description, reference information, or other fields for one or more of the array parameters. To do this, a 'parameter' command may be used after the array is specified, where the parameter has the same name as one of the arrays in the array table. The parameter item should not specify a value, but may specify a 'unit' for the parameter. The 'units' row is optional in the array since the array values may not have units, and since the units may be specified as part of the 'parameter' command if desired. Note that the use of a 'parameter' command after the array applies to two dimensional arrays as well. The following is a spreadsheet example array:

Table 2.4: One Dimensional Array Example

array	**Time**	**Level**	**Pressure**
units	**s**	**m**	**MPa**
	1	0.2	1.6
	1.5	0.3	1.7
	2	0.5	1.9
	2.5	0.7	2.0

If a description and reference is needed for the array parameters, then the following spreadsheet layout could be used (note that units can be included with the array or parameter command or excluded altogether if not needed):

Table 2.5: One Dimensional Array with Parameter Command

array	**Time**	**Level**			**Pressure**
	1	0.2			1.6
	1.5	0.3			1.7
	2	0.5			1.9
	2.5	0.7			2.0

Table 2.5: One Dimensional Array with Parameter Command

parameter	**name**	**units**	**ref**	**loc**	**desc**
	Time	s	Jrnl	pg 52	Simulation Time
	Level	m	Jrnl	pg 57	Water Level
	Pressure	MPa	Jrnl	pg 54	System Pressure

The format of a 2D array is ad follows:

Table 2.6: Two Dimensional Array Command

array	<array name>			...
	<y axis name>	<y axis val 1>	<y axis val 2>	...
	<x axis name>			...
	<x axis val 1>	<array val 1,1>	<array val 1,2>	...
	<x axis val 2>	<array val 2,1>	<array val 2,2>	...
	...	...	...	...

With a two dimensional array, the units for the array and for the axes must be specified using a 'parameter' command if units are required. The following is an example two dimensional array.

Table 2.7: Two Dimensional Array Example

array	Tmp			
	Yloc	0.2	0.4	0.6
	Xloc			
	0.1	95.2	99.3	103.7
	0.2	96.1	101	104.2
	0.3	97.4	102	105.6
parameter	name	units		
	Tmp	F		

Table 2.7: Two Dimensional Array Example

	Yloc	ft		
	Xloc	ft		

Note that arrays may be used in calculations, but when arrays are used in calculations, the equation is displayed symbolically, and will be shown with scalar values inserted, but array values will be shown symbolically. No final result is automatically printed. The results can be shown in a table however.

2.3.2 Param Naming Conventions

The parameters defined in the parameters database are intended to be useable in calculations in the calculation notebook. In order to use parameters in the calculation notebook, the parameters must use a name that is a valid Lua variable name. Lua variables are case sensitive and may include letters, numbers, and underscore characters, but may not start with a number.

Parameters also have to be represented in the Latex document, and thus need a valid representation in latex math mode syntax. The latex representation of a parameter can be specified explicitly in the parameters database by setting a 'tex' field for the parameter. However, a tex representation of the parameter name is autogenerated according to the following rules:

- The first underscore used in a parameter name represents a subscript. Each underscore thereafter is converted to a dot. For example, the name `VAR_121_x_23` is represented as $VAR_{121.x.23}$
- A range is specified in the subscript using the form NxN where the captital N values represent numbers. The 'x' value is converted to a dash. The range must come at the end of the subscript to be recognized as a range. For example, `VAR_121_1x10` is represented as $VAR_{121.1-10}$.
- Double underscores before the first underscore are represented as an underscore character. For example, `VAR__1__2_name` is represented as $VAR\_1\_2_{name}$

2.3.3 Adding Parameter Database Commands

There may be additional data items that it is useful to include in the parameter database. New commands can be added to support new data types. New commands can be registered in a 'QuARTICinit.lua' file stored in the folder with the executable or with the folder where QuARTIC is executed. This file is loaded before the parameters database is read.

A new command is registered using the `addCommand(name, fn)` function that is available from the `QuARTIC.TemplateContextLoader` package. The 'name' parameter is the command name and the 'fn' parameter is a function of the form `fn(cmds, row, info)` that recieves the commands parser,

the command parameter row, and an info parameter than indicates the line being processed. The command function will typically do the following:

- `cmds:getContext(name, default)` is called to get a table to store the commands that are read in. If no table has been registered previously, the default value is registered with the context and is returned. Otherwise, a previously registered table is returned. Note that the context is the template variable space, so the variables that are registered are available from the template file. In order to avoid conflicting with variable names that may be used in the template file, the standard is to start the variable name with an underscore.
- The `getCmdFieldReader` function from `QuARTIC.TemplateContextLoader` is called with the 'row' variable to get an object that is used process the command rows that follow. This object stored the field names and the columns that they are in, and includes a `mapRow2Fields` function that takes a command instance and gives back a table with named values pairs from the row that is processed.
- A command handler is registered that processes the command instances that follow the command header. This is registered as the 'HANDLER' value of the commands parser object (i.e., the first parameter passed to the registered command function). The handler function is also of the form `fn(cmds, row, info)`. As noted above, the row is typically processed in the handler function using the value returned from `getCmdFieldReader`.
- The row data is processed to generate an object and the object is stored in the table retrieved from `getContext`.

Below is a simple example of adding a new `test` command:

```
local tcl = require("QuARTIC.TemplateContextLoader")

-- Call addCommand to add a new command
tcl.addCommand("test", function(cmds, row, info)
  local test = cmds:getContext("_test", {})
  local fields = tcl.getCmdFieldReader(row)

  -- Register the handler function to process the command rows
  cmds.HANDLER = function(cmds, row, info)
    local vals = fields:mapRow2Fields(row)
    table.insert(test, vals)
  end
end)
```

The table that was registered via `getContext` is available from the template file and can be used to access any items that were added as part of the command. In this case, the items can be access from the `_test` variable. In this simple example, the row data is not processed to create an object. However, the handler can be used to do more sophisticated initialization of objects if needed.

2.4 Template Files

Template files are written in latex format. A brief discussion of latex is presented in Section 2.4.1.

2.4.1 Latex

Latex, which is pronounced as 'Lah-tek', is a document preparation system for technical documents that has been around for many years. Latex documents are written in plain text, and as a philosophy, latex tries to make it easy for authors to add content without having to focus too much on appearance. A document writer uses tags to identify section headings, and blank lines to indicate the end of a paragraph. However, aside from blank lines, whitespace is generally ignored.

If you are unfamiliar with latex, there are a number of resources available on the internet that provide assistance. You can search for "A Beginners Guide to Latex" by David Xiao, which provides a brief introduction to latex. This shows how to format a latex document, how to include sections, tables, lists, labels and cross references, and how to add equations or mathematical expressions. A more comprehensive introduction is the "Latex for Beginners" workbook. This covers the topics addressed in "A Beginners Guide to Latex". In addition, it shows how to add a table of contents, how to use font effects, such as bold or italics, how to adjust font color or size, how to add figures, and how to add a bibliography and reference bibliography items.

Latex has a basic set of features built in that are useful for writing technical documents. The basic set of features can be expanded or modified using latex packages that add additional capability. The site http://tug.ctan.org has a repository of latex packages, and documentation for the packages tends to be included. Note that texlive has many packages preinstalled, so that they are available for use without requiring a separate installation process. Summaries of commonly used latex packages can be found on the web. One such summary is located at https://en.wikibooks.org/w/index.php?title=LaTeX/Package_Reference.

2.4.2 Template File Syntax

Latex is a useful tool for writing technical documentation. However, it does not have a convenient way to perform live calculations (i.e., symbolic calculations that are automatically expanded based on values defined for the variables). In order to add this capability, a syntax was defined for embedding Lua scripting inside a latex document. Embedded scripting is useful in some other ways as well. For example, the parameters that are used in the document are tracked, and an autogenerated table of inputs with references can be inserted into the document sith a simple script command.

Script blocks are included in the document using special markers that identify the block of text as Lua code. A few different block types are supported:

Code Block: @[<Lua code>]@ or @ at start of line
This is used to include a block or line of Lua code in the latex document. For example, to

call the function `InputTable` one of the following can be used:
`@[ InputTable() ]@`
`@ InputTable()`

Equation Block: @$ <eqn> $@ or @$$ <eqn> $$@
This is used to include an inline equation or a standalone equation. The inline equation is typically included in within a paragraph, or sometimes a table cell, while the standard equation is on its own line in the document, and may be numbered.

Comment Block: @# #@ or # at start of line
A comment block or line is skipped and not included in the document.

Some of these blocks include modifiers or options that adjust the behavior. These are discussed in more detail, in the sections that follow, with examples of usage.

2.4.2.1 Code Blocks

Script blocks are used to directly include Lua code in a document. The code can be used for a wide variety of purposes, but on primary purpose is to write content to the latex document. Because this is so common, a special modifier has been included in the code block syntax. The code block syntax is:

```
@[-=? <Lua code> -]@       (Inline block)
@=? <line of Lua code>     (Single line block)
```

An inline block can be used to insert Lua code inside a paragraph. It can also be used to insert Lua code that spans multiple lines. The start and end markers allow flexibility in the placement of inline blocks. The block can start and end at any location within the document, except for within another block, and allows arbitrary Lua code to be included, within a few limitations. For example, the Lua code cannot contain ']' since this is used to close a block. There are also some predefined names that the block should not use.

The single line block only requires a marker at the start of the line (no spaces are allowed before the '@' symbol), but can only span a single line. This is useful for short commands, but is less flexible.

2.4.2.1.1 Code Block Modifiers The modifiers/options must come immediately after the opening marker ('@[' for multiline blocks or '@' for single line blocks) or immediately before the closing marker (']@'), with no spaces between.

The whitespace (-) modifier

The '-' modifiers for the multline block, which may be included with the opening or closing markers, cause any whitespace and newline before or after the text (depending on whether the minus is used with the opening or closing marker) to be removed from the document. The minus sign must be

placed immediately after the opening marker ('@[') or immediately before the closing marker (']@'). No other modifier can be placed between the (-) and the marker.

In general, latex is insensitive to spaces and newlines, in the sense that multiple spaces are treated as a single space when generating the pdf, and multiple blank lines are treated as a single blank line (marking the start of a new paragraph). Occasionally the removal of whitespace and newline will impact the format of the final PDF, but in general, it mostly impacts how the final 'tex' file looks. Removal of extra whitespace can make the document look neater and easier to read.

The write (=) modifier As noted previously, one of the most common purposes of including a block of Lua code is to write dynamically generated content (i.e., content that can change) to the latex document. The '=' modifier causes the item contained in the block to be written to the document. As an example, assuming a parameter $p = 5$ MPa, and assuming a variable $a = 100$, and a function 'add2' is defined that takes a value and adds 2, as in the following code block (which does not include a write modifier), the table that follows shows how these values can be printed.

```
@[
a = 100

function add2(x)
  return x + 2
end
]@
```

Block Syntax	Result	Description
`@[= p ]@`	$p = 5$ MPa	When a parameter is printed, the name of the parameter is included as well as the value and units.
`@[= a ]@`	100	For a standard Lua variable, the value of the variable is printed.
`@[= add2(5) ]@`	7	For a function, the value returned from the function is printed.

The delay (?) modifier

In some cases, a value may need to be printed in the document before it has been defined. In this case, the template cannot write the value to the output because the value is not known. The delay (?) modifier causes the item to be written out after the document has been fully processed. Thus the item can be printed even if it was defined after the request was made to write the item.

Note that this works well if the value of an item (i.e., variable or parameter) is only specified once. However, if the value changes in the document, the value that is printed may not be the desired value. Thus, it is recommended that this only be used with values that are defined once and not modified. In the example below, the value of b is not defined until after it is printed.

```
the value of b is @[=? b ]@.

@ b = 10
```

the value of b is 10.

2.4.2.2 Equation Blocks

Equation blocks are used to perform calculations in a calculation notebook and automatically document the equations. Two types of equation blocks are supported: inline equations (@$ <eqn>$@), which can be placed inside a paragraph or a table cell, and display equations (@$$ <eqns> $$@), which are represented in their own paragraph, and may include equation numbers. There are also a set of equation block modifiers that may come after the open tag marker, as shown in the syntax example below. The modifiers are discussed in Section 2.4.2.2.3.

```
@$>#^ <equation(s)> $@
@$$>1-^ <equation(s)> $$@
```

Equations within the equation blocks are written in standard Lua mathematical syntax. All variable names must be valid Lua variable names, although these will be rendered as indicated in Section 2.3.2. More than one equation can be included in an equation block, and when a set of display equations is grouped, it is recommended that these be included in a single block since spacing between equations is large when equations are included in separate equation blocks. For example, consider the equation blocks shown below and rendered thereafter:

```
@$$ L_1 = 5*m $$@
@$$ L_2 = 2*m $$@
@$$ L_3 = 1*m $$@
```

$$L_1 = 5 \text{ m} \tag{2.4.1}$$

$$L_2 = 2 \text{ m} \tag{2.4.2}$$

$$L_3 = 1 \text{ m} \tag{2.4.3}$$

When this is included in a single equation block, the spacing is better (note that the '-' modifier is applied which causes equation numbers to not be included for this repeat definition of the lengths):

```
@$$-
L_1 = 5*m
L_2 = 2*m
L_3 = 1*m
$$@
```

$$L_1 = 5 \text{ m}$$

$$L_2 = 2 \text{ m}$$

$$L_3 = 1 \text{ m}$$

Sometimes it is useful to include numeric calculations with raw numbers rather than parameters. When raw numbers are used, the value is computed before generating an equation. Thus, the calculation is not documented. For example, the equation block below produces only a final value of x as shown:

```
@$$
x = 12*4/3
$$@
```

$$x = 16 \tag{2.4.4}$$

To represent the actual calculation, the numbers need to be converted to parameters. Note that a number that is multipled by a parameter is automatically converted to a parameter, so it is adequate to convert 4 in the equation above to a parameter. The function `P_` is used to convert a number to a parameter as shown below:

```
@$$1
x = 12*P_(4)/3
$$@
```

$$x = \frac{12 \cdot 4}{3} = 16 \tag{2.4.5}$$

2.4.2.2.1 Equation Block Functions A few different functions are available to be used in equation blocks. The table below shows the built in functions.

Table 2.8: Equation Block Functions

Function	Description
`sin(angle)`	Calculate the sine of the angle. Note that the angle should be a parameter with units of radians or degrees.
`cos(angle)`	Calculate the cosine of the angle.
`tan(angle)`	Calculate the tangent of the angle
`asin(x)`	Calculate the inverse sine of x.
`acos(x)`	Calculate the inverse cosine of x.
`atan(x)`	Calculate the inverse tangent of x.
`exp(x)`	Calculate e^x.
`log(x, base)`	Calculate the log of x using the given base. The base parameter is optional with the default base being 'e' (i.e., natural log).

Below are examples of using these functions.

```
@$$<1
r_1 = sin(90*deg)
r_2 = cos(0*deg)
r_3 = tan(45*deg)
r_4 = asin(0.5)
r_5 = acos(0.5)
r_6 = atan(0.5)
r_7 = exp(-1)
r_8 = log(2)
$$@
```

$$r_1 = \sin(90^\circ) = 1 \tag{2.4.6}$$

$$r_2 = \cos(0^\circ) = 1 \tag{2.4.7}$$

$$r_3 = \tan(45^\circ) = 1 \tag{2.4.8}$$

$$r_4 = \text{asin}(0.5) = 30^\circ \tag{2.4.9}$$

$$r_5 = \text{acos}(0.5) = 60^\circ \tag{2.4.10}$$

$$r_6 = \text{atan}(0.5) = 26.565^\circ \tag{2.4.11}$$

$$r_7 = \exp(-1) = 0.36788 \tag{2.4.12}$$

$$r_8 = \log(2) = 0.69315 \tag{2.4.13}$$

2.4.2.2.2 Equation Blocks Create 'Param' Objects Equation blocks are special in that the variables calculated in an equation block is a Param object (see Section 2.4.10.3). Param objects are the basic component of symbolic calculations in QuARTIC®.

An essential characteristic of a Param object defined in an equation block is that it records its own variable name. For example, when the following code block is used:

```
@$$ A_100 = 120*m $$@
```

$$A_{100} = 120 \text{ m} \tag{2.4.14}$$

then the parameter name is stored as `A_100.name` which has the value `A_100`. To display a Param name in the document, the form `$@[=A_100.tex]@$` may be used, which in this displays: A_{100}.

When used in subsequent equations, a named Param object displays its symbolic name (i.e., 'tex' value) in the equation. Normal Lua variables, as defined in Lua code blocks, do not have access to a variable name, and will not show a symbolic name if use in equations.

When creating template functions that perform calculations, it can be useful to construct parameter names in an equation block. To do this, two functions are available:

- **__Param(name, value)** takes the parameter name as a string and a parameter value. The value can be a calculation, just as when setting a parameter using the equals symbol.
- **join(...)** is a function that takes a set of values, which can be strings or numbers and appends them together to form a string. The string that is formed should be chosen to be a valid variable name.

The parameter that is created can be assign to a variable. However, the variable in this case will be a temporary variable that is only available in the context of the equation block. Whenever the variable is used in an equation, the name provided to the `_Param` function will be shown in the equations. Thus, the assigned variable is a temporary alias for the parameter as exhibited in the example below.

```
@ a = 5
@$$1
b = _Param(join("b_",a,"_x"),   15*m)
_Param(join("b_",a+1,"_x"), 10*m)
_Param(join("c_",a,"_x"), b/5)
$$@
```

$$b_{5.x} = 15 \text{ m} \tag{2.4.15}$$

$$b_{6.x} = 10 \text{ m} \tag{2.4.16}$$

$$c_{5.x} = \frac{b_{5.x}}{5} = \frac{15 \text{ m}}{5} = 3 \text{ m} \tag{2.4.17}$$

2.4.2.2.3 Equation Block Modifiers The modifiers (i.e., >, #, ^, <, -, and 1) are all optional, and can be used alone or in combination with other modifiers. The ordering does not matter. The purpose of each modifier is described below.

The Output Parameter Modifier (>)

The output modifier applies to both the inline and display equation blocks. This modifier marks a parameter as an output parameter. Output parameters are written to a csv file when the calculation notebook is processed. In the following, the parameters a, b, and c are marked as output parameters:

```
@$$>
a = 10*s
b = 20*s
c = 30*s
$$@
```

Show Numerical Value Only Modifier (#)

The show numerical value only modifier applies to the inline equation block. Typically, when an equation block is included, the name of the variable is shown, followed by the symbolic calculation, and then the numerical result as shown in the example below:

```
@$ V = A*L $@
```

$$V = A \cdot L = 2 \text{ m}^2 \cdot 5 \text{ m} = 10 \text{ m}^3$$

However, in some cases, it may be preferrable to show only the result. One common use for this is showing calculated values in a table, where the name pattern for the calculated values may be shown in the heading of the table. For example, a table might be used to calculate the volume from length and area. Suppose the following parameters are defined: $A_1 = 2 \text{ m}^2$, $A_2 = 2 \text{ m}^2$, $A_3 = 2 \text{ m}^2$, $L_1 = 3 \text{ m}$, $L_2 = 6 \text{ m}$ and $L_3 = 5 \text{ m}$. A table is included that calculates the volume in the last column using the equation below (where i is replaced by the specified index):

```
@$# V_i = A_i*L_i $@
```

i	A_i	L_i	V_i
1	2 m^2	3 m	6 m^3
2	2 m^2	6 m	12 m^3
3	2 m^2	5 m	10 m^3

No Numerical Substitution Equation Modifier (<)

This causes the numerical substitution display of the equation to be suppressed. This is particularly useful when QuARTIC fails to recognize that an equation has no symbolic variables. In this case, the numerical equation can be printed twice, so this option can be used to suppress the exteraneous printed equation.

```
@$$<
x = sin(30*deg)/2
$$@
```

$$x = \frac{\sin\left(30^\circ\right)}{2} = 0.25 \tag{2.4.18}$$

The Single Line Equation Modifier (1)

When a display equation block is included in a document, then the equation is typically represented on 3 lines. The first line shows the symbolic representation of the equation. The second line shows the equation with values inserted, and the third line shows the final value. Sometimes all three of these will fit on a single line. The 1 modifier causes the equations within the block to be represented on a single line. For example, suppose that $x = 2$ m and $y = 2$ m. The following equation is relatively short, but gets spread across 3 lines:

```
@$$ r = (x^2 + y^2)^0.5 $$@
```

$$\begin{aligned} r &= \left(x^2 + y^2\right)^{0.5} \\ &= \left((2\text{ m})^2 + (2\text{ m})^2\right)^{0.5} \\ &= 2.8284\text{ m} \end{aligned} \tag{2.4.19}$$

The equation is represented more compactly with the '1' modifier.

```
@$$1 r = (x^2 + y^2)^0.5 $$@
```

$$r = \left(x^2 + y^2\right)^{0.5} = \left((2\text{ m})^2 + (2\text{ m})^2\right)^{0.5} = 2.8284\text{ m} \tag{2.4.20}$$

The No Equation Label Modifier (-)

By default, display equations include an equation number. This is useful if one wishes to reference the equation in another part of the document. However, some equations may be local and temporary in nature, and it may be desirable to exclude an equation number. The '-' modifier eliminates the equation number from equations contained in the block. For example the following equations include no equation number:

```
@$$-
x = 5
y = 3
$$@
```

$$x = 5$$

$$y = 3$$

The Hide Equation Modifier (^)

The hidden equation modifier causes the equations in the block to be calculated, but no result is shown in the document. The -H command line option causes hidden equations in the document to be displayed.

2.4.2.2.4 Equation Style In some cases, it is convenient to override equation modifiers. For example, a long equation might be included in a block of short equations. It makes sense to use the single line equation modifier (1). But it might be necessary to override this for one (or more) equations. Another example, is a block of equations where all the equations are output parameters (>) with exception of one to two equations. It is convenient to be able to modify the behavior on these equations.

Given a variable `param`, the following methods are available to override the modifiers:

- '`param.save = false`' indicates that are parameter is not an output parameter and overrides the output parameter modifier (>). This must come within the equation block AFTER the parameter equation. This is not required to come immediately after, but must be within the same equation block.
- '`param.save = true`' marks the parameter as an output parameter when the output parameter modifier is NOT used. This is useful if an output parameter is included in a block of multiple equations that are not output parameters. This must come in the equation block after the parameter equation.
- '`param.style.one = false`' will mark the equation as a multiline equation. This will override the single line equation modifier (1). This must be included after the parameter equation.

- '`param.style.one = true`' will mark the equation as a single line equation when the single line equation modifier is not used. This must be specified after the parameter equation.
- '`param.style.after = [[latex]]`' is used to change the latex that comes after each equation. By default the command `EqnSkip` is inserted, which is used to insert negative vertical space so that equations are grouped closer together. Note that the `EqnSkip` command should be defined at the top of the latex document.
- '`param.style.nolabel = true`' is used to make the equation unlabeled, unnumbered, and disables saving of the parameter.

The following can be set to modify the value for all equations within the equation block that follow the given command. This will not affect parameters where the value is set explicitly for the parameter.

- '`style.one = value`'
- '`style.save = value`'
- '`style.after = [[latex]]`'

The following is an example of the one property with the single line property disabled for the last equation:

```
@$$1-
R  = 10*m  -- Cylindircal Annulus Outer Radius
Th = 0.5*m -- Anulus Thickess
r  = R-Th  -- Annulus Inner Radius
h  = 20*m  -- Cylindrical Annulus height
Vol = h*Pi*(R^2 - r^2)  -- Volume of Cylindrical Annulus
Vol.style.one = false
SA  = h*2*Pi*(R+r) + 2*pi*(R^2-r^2) -- Surface Area
SA.style.one = false
$$@
```

OR

```
@$$1-
R  = 10*m  -- Cylindrical Annulus Outer Radius
Th = 0.5*m -- Anulus Thickess
r  = R-Th  -- Annulus Inner Radius
h  = 20*m  -- Cylindrical Annulus height
style.one = false
Vol = h*Pi*(R^2 - r^2)  -- Volume of Cylindrical Annulus
SA  = h*2*Pi*(R+r) + 2*pi*(R^2-r^2) -- Surface Area
$$@
```

$$\pi = 3.1415$$

$$R = 10 \text{ m}$$

$$Th = 0.5 \text{ m}$$

$$r = R - Th = 10 \text{ m} - 0.5 \text{ m} = 9.5 \text{ m}$$

$$h = 20 \text{ m}$$

$$\begin{aligned} Vol &= h \cdot \pi \cdot \left(R^2 - r^2\right) \\ &= 20 \text{ m} \cdot 3.1415 \cdot \left((10 \text{ m})^2 - (9.5 \text{ m})^2\right) \\ &= 612.59 \text{ m}^3 \end{aligned}$$

$$\begin{aligned} SA &= h \cdot 2 \cdot \pi \cdot (R + r) + 2 \cdot \pi \cdot \left(R^2 - r^2\right) \\ &= 20 \text{ m} \cdot 2 \cdot 3.1415 \cdot (10 \text{ m} + 9.5 \text{ m}) + 2 \cdot 3.1415 \cdot \left((10 \text{ m})^2 - (9.5 \text{ m})^2\right) \\ &= 2511.6 \text{ m}^2 \end{aligned}$$

2.4.2.2.5 Inserting latex between equations To insert arbitrary latex between equations, the `latex([[<latex to insert>]])` command may be used within the equation block. Note that if this is done, the default latex placed after the equation may need to be changed for formatting to work correctly.

```
@$$1-
a = 14
a.style.after = [[]]
latex([[Comment between equations.
Note that this can be multiline.]])
b = 12
$$@
```

$$a = 14$$

Comment between equations. Note that this can be multiline.

2.4.2.2.6 Explicit Parenthesis In some cases, it may be useful to control the placement of parenthesis within equations. A special `paren(style)` command may be used to explicitly add parenthesis to an equation. The type of parenthesis may also be specified as a parameter passed to `paren`:

- '()' (in quotes) causes standard parenthesis to be used. This is the default option, so including this is not necessary.

- '[]' (in quotes) causes square brackets to be used.
- '{}' (in quotes) causes curly braces to be used.

The following is an example:

```
@$$-
L_1 = 5*m
L_2 = 6*m
L_3 = 3*m
Th_1 = 0.1*m
Th_2 = 0.2*m
Th_3 = 0.3*m
L_tot = (L_1 + 2*Th_1):paren() + (L_2 + 2*Th_2):paren("[]") + (L_3 +
    2*Th_3):paren("{}")
$$@
```

$$L_1 = 5 \text{ m}$$

$$L_2 = 6 \text{ m}$$

$$L_3 = 3 \text{ m}$$

$$Th_1 = 0.1 \text{ m}$$

$$Th_2 = 0.2 \text{ m}$$

$$Th_3 = 0.3 \text{ m}$$

$$\begin{aligned} L_{tot} &= (L_1 + 2 \cdot Th_1) + [L_2 + 2 \cdot Th_2] + \{L_3 + 2 \cdot Th_3\} \\ &= (5 \text{ m} + 2 \cdot 0.1 \text{ m}) + [6 \text{ m} + 2 \cdot 0.2 \text{ m}] + \{3 \text{ m} + 2 \cdot 0.3 \text{ m}\} \\ &= 15.2 \text{ m} \end{aligned}$$

2.4.2.2.7 Handling Long Equations Sometimes equations are quite long, and need to be split across multiple lines when they are shown in the document. A special `brk(mode)` command is available for use in equations to specify where breaks are to occur. The `mode` parameter may have the values:

- 'symbolic' (in quotes) to indicate that only the symbolic equation should break to the next line at the location. 'tex' can be used as a shorthand alternative to 'symbolic'.
- 'value' (in quotes) to indicate that only the equation with numerical values inserted should break to the next line at this location.
- No value to indicate that the equation should break for both the symbolic and numerical equation.

Note that there are some places within an equation where breaks are not allowed (such as within a parenthesized block, or within terms contained in a fraction). Placing a break in such locations will cause errors when trying to convert the latex file to a PDF document. Below is an example of

an equation with a long symbolic representation. Thus, the symbolic representation needs to have a break. The equation with valies substituted fits on a single line, so only the symbolic equation includes a break.

```
@$$-
Length_x_101 = 100*m
Length_x_102 = 50*m
Length_x_103 = 60*m
Length_x_104 = 80*m
Length_x_105 = 120*m
Length_x_106 = 50*m
Length_total = (Length_x_101 + Length_x_102 + Length_x_103
               + Length_x_104 + Length_x_105):brk('symbolic') + Length_x_106
$$@
```

$$
\begin{aligned}
Length_{x.101} &= 100\ \text{m} \\
Length_{x.102} &= 50\ \text{m} \\
Length_{x.103} &= 60\ \text{m} \\
Length_{x.104} &= 80\ \text{m} \\
Length_{x.105} &= 120\ \text{m} \\
Length_{x.106} &= 50\ \text{m} \\
Length_{total} &= Length_{x.101} + Length_{x.102} + Length_{x.103} + Length_{x.104} + Length_{x.105} \\
&\quad + Length_{x.106} \\
&= 100\ \text{m} + 50\ \text{m} + 60\ \text{m} + 80\ \text{m} + 120\ \text{m} + 50\ \text{m} \\
&= 460\ \text{m}
\end{aligned}
$$

2.4.2.3 Comment Blocks

Template comment blocks are text that is excluded from the final Latex document.

```
#  A '#' symbol at the start of the line indicates a comment line
@# This is a comment block, which can span multiple lines. #@
```

2.4.3 Including External Template Files

In some cases, it is useful to break documentation into multiple files. QuARTIC® has the ability to include external template files in a template file. This can be useful for keeping the size of

files manageable for large documents. As a simple example, consider a file called 'test.tmp' that contains the following text:

```
This text comes from 'test.tmp'.

@$$ x=1 $$@
```

The template file can be included via the 'include' function as follows, with the result shown afterward:

```
@[ include("test.tmp") ]@
```

This text comes from 'test.tmp'.

$$x = 1 \tag{2.4.21}$$

2.4.4 Dynamic Template Functions

The template file may contain sections that have a standard format, but document different data. QuARTIC® has the ability to include functions which can be used to make portions of the document into macros that are expanded with data that is passed to the function. This can be used for simple parts of a document that are repeated frequently, or for large and complex sections. Note that macro functions can be included in the base document, or as external template files that act like libraries which may be useful in multiple documents.

As a simple example, consider the following function which receives a variable which is a list and generates an enumerated list from the items:

```
@ function makeList(items)
Here are the list items:

\begin{enumerate}
@ for i, item in ipairs(items) do
  \item @[= item]@
@ end
\end{enumerate}
@ end
```

This can be called with a list of items as follows with the results shown afterward:

```
@ makeList({'First item', 'Second item', 'Third item','And so on'})
```

Here are the list items:

1. First item
2. Second item
3. Third item
4. And so on

2.4.5 Including Hooks

For more complex sections that are rendered as macros, it is common to need to insert customized content provided by the user. In order to accomodate this, a function `try(fn_name, ...)` is available that takes the name of a function to call, and a set of parameters to pass to the function. Inclusion of the function in the document is optional. A function like this that is optionally included is referred to as a **hook**. If the hook with the given name is present in the document, it is called with the extra parameters passed to `try`. The hook may either write directly to the document, or may return a string which is written to the document. It may be convenient to contruct the hook name at runtime and include values such as component number as part of the hook name.

If macros become large and try to call multiple functions, it is useful to be able to display potential hooks in the document to make it clear where content can be inserted. The following parameter is used to cause the hook functions to be shown in the document in red to make it clear where additional information can be inserted.

Variable	Description
`ShowTry`	If this is set to `true` hook functions will be shown in the document at the location where the content will be added if the hook is present. The hooks are not shown if the value is `false` or `ShowTry` is not present. Only hook functions that are not present in the document are show. If `ShowTry` is set to `"verbose"`, then annotation is shown around the content that is inserted by hooks that are in the document as well.

Consider an example case. The input for a pipe component for an analysis code is being documented. A macro called 'PipeDoc' is created that is passed a data structure with pipe inputs. An input called 'ID' is included that gives a pipe ID number, and an input called 'length' is available that is an array of pipe segment lengths in meters which will be shown in a table. The function might look something like the following (note that indentation is included for readability, but is not required in actual document):

```
@ function PipeDoc(cmp)
  @[ try("PipeLen" .. cmp.ID) ]@

  For pipe number @[= cmp.ID]@, the cell lengths are:

  \begin{tabular} {c c}
    \hline
    Cell \# & Length (m) \\
    \hline
    @# LOOP THROUGH THE CELL LENGTHS #@
    @[ for i, length in ipairs(cmp.length) do ]@
      @[=i]@  & @[=length]@ \\
    @[ end ]@
  \end{tabular}

  [Additional pipe info documented here]
@ end
```

If the function is called, and the callback macro is not defined, then the callback will be skipped and the rest of the content wil be rendered as shown below:

```
@ PipeDoc({ID=2, length={1,3,2,4}})
```

For pipe number 2, the cell lengths are:

Cell #	Length (m)
1	1
2	3
3	2
4	4

[Additional pipe info documented here]

If the function 'PipeLen2' is defined before 'PipeDoc' for component 2 is called, then the callback function will be called and included in the documentation as shown below:

```
@ function PipeLen2()
  For pipe 2, the pipe lengths are calculated as:

  @$$
  L2_1 = 1
  L2_2 = 3
  L2_3 = 2
  L2_4 = 4
  $$@
@ end

@ PipeDoc({ID=2, length={1,3,2,4}})
```

For pipe 2, the pipe lengths are calculated as:

$$L2_1 = 1 \tag{2.4.22}$$

$$L2_2 = 3 \tag{2.4.23}$$

$$L2_3 = 2 \tag{2.4.24}$$

$$L2_4 = 4 \tag{2.4.25}$$

For pipe number 2, the cell lengths are:

Cell #	Length (m)
1	1
2	3
3	2
4	4

[Additional pipe info documented here]

2.4.6 Lua Functions Available in Template File

As with standard Lua, a template can access function in an external lua file by using the `require` command to load the module. However, a few functions, shown in the table below, are available by default.

Table 2.9: Functions Available by Default

Function	Description
`addFigure(name, opt)`	This command is used to add a Figure that thas been specified in the parameter database via the `figure` command to the document. The `name` must match the name specified for the figure in the parameter database. The `opt` parameter is an optional table that can include the following options. These options override values specified in the parameter database: `caption` the figure caption. `width` the figure width. `height` the figure height. `label` the label to use for referencing the figure. `pos` set the placement specifier. The default value is "H", which requires the `float` latex package to be included. `figureFolder` the folder where the figure is located. `iscapbelow` boolean to indicate if caption is below (or above the figure) - true or false (default is true). Note that the default value for `iscapbelow` can be changed via the `iscapbelow` value in the module `tex.Figures`.
`dbg()`	Start a debug session at the current point in the template. A debug session is used to step through the template code to diagnose unexpected results.

Continued on next page

Table 2.9: Functions Available by Default (Continued)

Function	Description
`INCLUDE(template)`	Latex includes an `\include` command that is used to insert an external latex file document into the document. QuARTIC allows template files to be defined that that need to be processed. `INPUT(template)` is a QuARTIC command that processes the given template file and adds it to the output folder. It then includes a Latex `\include{<latex file>}` in the document. A Latex include command causes the included latex file to be processed separately. If a document is divided into separate files and is added via the include command, this can speed up processing of the document since only modified documents have to be processed. However, this also cause a new page before and after the included document, which will only work at certain locations in the document. Note that INCLUDE can only be used in the top level file. You cannot include a file in another file that was included or Latex will fail. Also note that the name of the template is case sensitive. If the template is included in a subfolder of the template directory, then folder names are also case sensitive and '/' should be used as the path separator rather than '\'.
`INPUT(template)`	Latex includes an `\input` command that is used to insert an external latex file into the document. This is similar to the INCLUDE command above, but uses the Latex `\input` command instead. This causes the external latex file to be added to the document at the location specified. It does not cause the documents to be processed independently by Latex. Thus, there is no speed up of processing. However, it also does not cause a page break before and after the content. This is a good way to partition a large document into separate files.
`isParam(var)`	Check whether a variable is a `Param` object. Param objects are specified in an external input file or are generated via equations or may be created with the `Param` method below.
`isUnit(var)`	Check whether a variable is a Unit object. Several unit variables are predefined and can be used in equations to get unit consistency.

Continued on next page

Table 2.9: Functions Available by Default (Continued)

Function	Description
`latexEsc(text)`	Write a string as latex, converting special symbols so that they print in latex. For example, the symbols $, {, }, [, and] are converted to a form where they are printable.
`Param(val)`	Create a parameter outside an equation block. The *val* parameter can be a number to indicate the parameter value, a string that indicates the parameter name, or a table that contains *name* and *value*, which provide the variable name and its value (i.e., `Param({name="h", value=14.7})`). A *tex* value can also be included, which is a user defined representation of the variable, if the default representation is not adequate. If *val* is a string, a second parameter can be passed in that is the value (i.e., `Param("v",15)`).
`util.basename(filename)`	Get the base filename from a path (i.e., the name of the file without the path)
`util.copyTbl(tbl,opt)`	Copy a table. This function does a deep copy of a table. The opt table allows the option 'table_refs' to be specified as true. When this is true, tables are tracked as they are copied. If the same table is included mulitple times in the original table, then the same table is returned if 'table_refs' is true so that the tables remain consistent in the copied version.
`util.fileExists(path)`	Check if t he file indicated by the given path exists. This returns the file type. If the file does not exist, it returns `nil`, which equates to false.
`util.filter(fn,tbl,ctxt)`	Return a table with values from 'tbl' such that `fn(val, i)` returns a true value. If an initial 'ctxt' table is passed in, items are appended to this table.
`util.firstFn(fn,tbl,start)`	Starting at the given 'start' index (default 1), call `result = fn(val, i)` and return val, i, and result for the first result that does not evaluate as false.

Continued on next page

Table 2.9: Functions Available by Default (Continued)

Function	Description
`util.foreach(fn,tbl,ctxt)`	This loops over each item in the table (tbl) and calls the provided function which is called as `fn(val,i,ctxt)`, where 'val' is the table value, i is the table index, and 'ctxt' is the third parameter passed to foreach. If 'tbl' is an integer value, then it loops over values from 1 to 'tbl' and the function has the form `fn(i,ctxt)`. If no 'ctxt' value is passed to foreach, a new table is created. The return value from `foreach` is the 'ctxt' value, so any intended results should be added to this.
`util.foreachpair(fn,tb,ctxt)`	Similar to foreach, but loops over key value pairs in a table. The function is called as `fn(val,key,ctxt)`, where 'val' is the table value, and 'key' is the table key. 'ctxt' is the same as for foreach.
`util.hasField(tbl, field)`	Check if the given table contains a key indicated by field.
`util.insert(tbl, i,` item)	Insert an item into the table at the given index and return the table.
`util.insertUnique(tbl,value)`	Insert the value into the tbl if the value is not already in the table.
`util.inSet(set, item)`	Given a table of items, check if the given item is in the table. If so return the item. Otherwise return `nil`.
`util.isInt(num)`	Checks if a numeric value is an integer.
`util.isAfterVersion(x,x,x)`	Check if the QuARTIC version is after the given version number. For example, `util.isAfterVersion(1, 1, 16)` checks to see if the QuARTIC version being used is after version 1.1.16.
`util.joinTbls`	The function takes an arbitrary number of parameters that can be tables or values and joins them together into a single table. For tables it adds each item from the table to the joined table. For individual values, it adds the value to the table. The joined table is returned.
`util.map(fn,tbl)`	Call `fn(val, i)` for each value in the table, where 'i' is the value index. Add the return value to a results table and return the result table. If the `fn` returns `util.SKIP`, skip adding the value to the table. If a positive integer is passed in as 'tbl', loop over one to the integer value and call `fn(i)`. Return the results table.

Continued on next page

Table 2.9: Functions Available by Default (Continued)

Function	Description
`util.range(start,stop,step)`	This creates a table with values starting at the start value, incrementing by the step up through the stop value.
`util.reduce(fn,tbl,init)`	For each item in the table and starting with init as the first result, call the function `result = fn(result, item, i)`. Return the final result.
`util.removeIndexes(tbl, indexes)`	Remove the indexes in the given list from the table and return the table.
`util.split(str,ptn,fmt)`	This function is used to split a string. It takes the string (str), a split pattern (ptn) to match, and a function `fmt(part)` that is passed each split part of the string and returns a value to add to the result table.
`util.splitext(filename)`	Split a filename into the root name and the extension, where the extension starts with a dot.
`util.splitpath(path)`	Split the path into two parts - the path up to the last separator, and the folder or filename after the last separator.
`util.tableContains(tbl, item)`	Check if the table contains the given value. Return the index of the item if it is in the table. The item can be a value or a function. If the item is a function, the function is called with the table value and returns true if the value is the one that is sought for.
`util.tableKeys(tbl,fn,int)`	Get the table keys from the given 'tbl'. Sort the values using the given comparison `fn(a,b)`. By default integer keys are not included. If 'int' is true, then integer values will be included among the keys.
`util.trim(str)`	Trim the whitespace from around a string.
`util.unpackTbl(tbl,keys,default)`	Extract and return the items associated with the specified keys. If the item is not present, use the given default value.

2.4.7 Event Listener Functions

There are a few events that are handled by creating a specific listener function. These events allow you to override default QuARTIC behavior.

2.4.7.1 The `onSaveParam` Event

The `onSaveParam` event is called after the document is processed as the output parameters are being saved. The function form is `onSaveParam(type, csv, param, hasValue)`. There are two types that are possible: "header" and "row". The header is passed as the first call and provides the csv header string for the data that is written. In this case, no `param` or `hasValue` parameter is passed to the function. The function may return an alternative header string to use.

For each row of the output file, the `onSaveParam` function is called with "row" as the type. In this case, the default csv is passed to the function along with the parameter and a flag indicating if the parameter could be evaluated to a value. If a value is returned, the value is written to the output file.

2.4.7.2 The `onValue` Event

The `onValue` event is called when values are printed in the document. This function is called as `onValueparam, fmt`, where param is the calculated parameter and fmt is a format string that can be used to print the value. If a value is returned from the `onValue` function, then this value is printed to the document. Otherwise the default value of `string.format(fmt, param._valstr_)` is printed to the document.

2.4.8 User Defined Modules

Modules can be loaded from a QuARTIC template. The module can be included in the folder with the QuARTIC executable or in a subfolder called 'lua'. It can also be included in the folder where QuARTIC is executed. In general the modules should be constructed as standard Lua modules. A standard Lua module is constructed in the form shown below:

```
local M = {}

function M.myfunc(...)
   -- Function logic
end

return M
```

Here a module table 'M' is defined. The name is arbitrary. Functions, and possibly other variables, are added to the module. The module is then returned. Module can include any number of module functions and variables as needed. The module is then requested using the 'require' function as shown below:

```
mymod = require("<module name=>")
```

The value that is returned from the call to 'require' is the module table that was returned from the module file. Any functions contained in the module can then be accessed and called from the variable returned from 'require'.

2.4.9 Autoloaded modules

There are a few modules that are autoloaded if present at specific times during initialization. The 'QuARTICinit' module was already discussed in Section 2.3.3, since this can be used to define new parameters database commands. Each of these modules must be included in the Lua path as indicated in the parent section, and must have the module name with a 'lua' extension. None of these modules are required. There is no need to return a table from these modules since the return value is not used. In essence, these modules can simply be treated as Lua scripts that are executed at specific times during initialization. The autoloaded modules are:

- `QuARTICboot` is requested first. At this point, the command line parameters are available via the `arg` table. These can be manipulated if desired at this point. The command line parameters have not been processed, so templates and parameter inputs have not been loaded.
- `QuARTICinit` is requested shortly after `QuARTICboot`. Again, this is before command line arguments are processed. For the most part, these could be used interchangeably.
- `QuARTICcontext` is called after the template context has been created. The template context is essentially the variable space the template uses. All variables that are available to the template are present in the context. New parameters can be added to this context at this point if desired.

The template context is available via the global variable `_tmpctxt`. Note that this can be accessed from the 'QuARTICcontext' module, but not from 'QuARTICboot' or 'QuARTICinit'. The context can also be accessed from user defined modules that are called from the template. As noted above, all variables accessible from the template are stored in the template context table. For example, the functions indicated in Section 2.4.6 are contained in the template context.

2.4.10 Lua Objects

This section discusses Lua objects that are specifically created for use in QuARTIC®.

2.4.10.1 _acronyms Object

The `_acronyms` object is populated with acronyms that are included in the parameter database. Recall that functions are called using `_acronyms:fn_name(params)`, where a colon is used before the function name. The following functions are defined for the `_acronyms` object.

Table 2.10: Builtin Events

Function	Description
`acronym(name, def)`	This function can be used to add a new acronym with its definition within the document. It is recommended that all acronyms be included in the parameter database, but this is not required. The acronymns is automatically marked as being in use in the document.
`addAcronyn(name, def, inUse)`	This is used to add an acronym to the acronyms list. The name, definition, and an in use flag are passed in. The in use parameter is optional and defaults to false.
`inUse(acronyms)`	This takes an acronym name or a table of acronyms and marks each as being in use in the document.
`sort()`	This alphabetically sorts the in use acronyms. Note that the in use acronyms are stored as array elements in the `_acronyms` table. Thus, `#_acronyms` is the number of in use acronyms.
`foreach(fn, ctxt)`	This is passed a function of the form `fn(acronym, def, idx, ctxt)` and returns whatever is included in the `ctxt` variable at the end.
`acronymsTable()`	This autogenerates generates a latex longtable with acronymns and definitions for each acronyms that is marked in use. This should be called as a delayed function that is processed after the document is completed.

2.4.10.2 _event Object

QuARTIC has a builtin event manager object that is used to listen for events or publish events. An event is a message that indicates something of interest has happened, and a listener is a function that is called when the event occurs. The listener function is passed the event name followed by any parameters that are included with the event notification.

Table 2.11: Builtin Events

Event	Description
'new template'	The `_tmpctxt` is created. This is passed as a parameter to listeners. This event occurs before the template is loaded.
'param defined'	This event is called when a parameter is created from the parameter database. The parameter is passed to the listener function. This occurs before the template file is loaded.

Continued on next page

Table 2.11: Builtin Events (Continued)

Event	Description
'new param'	This event is called when a new parameter is created. The parameter is passed to the listener functions.
'get param'	This event is called when a parameter is requested. The requested parameter is passed to the listener function
'template finished'	Called after the template file has finished executing
'on exit'	Called after QuARTIC has finished generating the document and before it exits.

The following functions are defined for events.

Table 2.12: Builtin Events

Function	Description
`_event:notify(event, ...)`	This is called to send an event notification. The 'event' is a string that uniquely identifies the event type. Any number of parameters can be sent with the event notification and these parameters are forwarded to the event listeners. Of course the parameter list should always be consistent for a particular event.
`_event:addListener(event, fn)`	This is used to register a new event listener. A listener function has the form `fn(event, ...)`, where 'event' is the event name, and ... represents the set of parameters passed by the `notify` function. The function is called each time an event notification is sent.
`_event:removeListener(event, fn)`	This is used to remove a registered event listener. Once the listener is removed, it will not be called when event notifications are sent.
`_event:clearListeners(event)`	Remove all the listeners for a particular event and return the list of listeners that was removed.

Below is a simple example of adding a listener, sending an event and removing a listener.

```
function myListener(event, param)
  print(param)
  -- When the listener is called, remove it from the list of listeners.
  -- Thus, it will be called only once.
  _event:removeListener("test event", myListener)
end

_event:addListener("test event", myListener)

_event:notify("test event", "A parameter to pass")
```

2.4.10.3 Param Object

Param objects are the basic building blocks of symbolic calculation in QuARTIC®. When a parameters database is loaded from a set of parameters database files (Section 2.3), the parameter items specified in the parameters database are converted to Param objects which can then be used in equation blocks (Section 2.4.2.2) in the template file(s) (Section 2.4). As new parameters are calculated in equation blocks, these can also be used in calculations in the document.

Param may have any of the properties specified for the `parameter` command in Section 2.3. Not all of these are required and not all are meant for direct use in a document. Some derived properties are defined that are intended use instead. Consider the following example parameter with value '10.12345' (4 signifiant digits are kept):

$$P_i = 10.12 \text{ MPa} \tag{2.4.26}$$

Below is a list of the Param properties and methods with a description and usage example:

Table 2.13: Param Object Properties and Methods

Name	Usage/Result	Description
cite	`@[=P_i.cite]@` Equation (2.4.26)	Provide a reference for the specified item. This may be a bibliography reference or an equation reference.
citeloc	`@[=P_i.citeloc]@` Equation (2.4.26) on page 41	Provide a reference for the specified item with the location.
convert2	`@[=P_i:convert2("kPa")]@` $P_i = 10123$ kPa	Convert the value to the units provided. The units can be a string or a units object. By default this function returns a new param object with the new units. A second parameter named *save* may be passed in. If the second parameter is *true*, then the units for the original parameter are modified.

Continued on next page

Table 2.13: Param Object Properties and Methods (Continued)

Name	Usage/Result	Description
fmt	`@[ P_i.fmt = "%0.2f" ]@`	This sets the format of a parameter based on the Lua `string.format()` command. If the format is not set explicitly, a default format will be used. The 'fmt' value is typically not printed in the document. This may also be used in an equation block after the variable has been assigned a value via an equation.
hasUnits	`@[=P_i:hasUnits()]@` true or false	Indicate whether the parameter has associated units.
num	`@[=P_i.num]@` 10.12345	Get the numerical value of the param object without units. Since this is a numerical value, the value will not be formatted if 'P_i.num' is printed.
numstr	`@[=P_i.numstr]@` 10.123	Show the formatted numerical value without units.
save	`@[ P_i.save = true ]@`	Marker that indicates that a parameter will be used in an external file (i.e., a TRACE input file).
show	`@[=P_i.show]@` $P_i = 10.123$ MPa	Show the variable and the associated value with units. Note that the value is automatically encapsulated in $ characters.
tblciteloc	`@[=P_i.tblciteloc]@` Equation (2.4.26) on page 41	Display the citation and location in a table cell. Break the citation and location into separate lines (for compactness).
tex	`$@[=P_i.tex]@$` P_i	Displays the latex representation of the variable. Note that $ is required around the value.
texstr	`@[=P_i.texstr]@` P_i	Same as the 'tex' value, but automatically surrounds value with $.
units	`@[=P_i.units]@` MPa	Show the param units. Parameters are not required to have units however.
val	`$@[=P_i.val]@$` 10123 kPa	Get the units object (value and units). The $ equation marker must be included around parameters with units.
valstr	`@[=P_i.valstr]@` 10.123 MPa	Print the formatted value and units as a formatted string. $ is automatically added to the printed value.

2.4.11 QuARTIC Modules

There are a number of modules available in QuARTIC that may be required inside a document. For some of these modules, code is make available with the QuARTIC installation that can be modified if needed to update the behavior. The following sections document modules that are available.

2.4.11.1 RegisterUnits Module

The `RegisterUnits` module should be included in the same folder as the QuARTIC executable. This module defines the basic unit types that are supported by QuARTIC as well as derived units. Basic units are registed in the function `M.RegisterBasicUnits`. For each basic unit, the function `BasicUnit(var, desc, tex)` is called, where `var` is the representation of the unit as a variable, `desc` is a description of the unit type, and `tex` is an optional latex representation of the unit. By default, the latex representation is the value of the `var` parameter.

The function `M.registerUnits` includes derived units that are based on the basic units that have been defined. These can simply be scaled versions of the basic units, such as deriving cm from m. Or they can define a new derived type, such as defining volume as `m^3`. In this case, the 'setkind(eqn, desc)' may be called to specify a type description for the new derived type. For example, `N = setkind(kg*m/s^2, "Force")` defines N (Newtons) as a unit of force. Only one instance of a new type should use `setkind`. Other scaled types should be scaled relative to the first instance.

2.4.11.2 InputTable Module

The `InputTable` module is used to track parameters that are used in a section and autogenerate a reference table that includes all parameters used in the section. The parameters come from the parameter database and should include a 'ref' field that gives the id of one of the references from the parameter database. It should also generally include a 'loc' parameter that indicates the specific location within the reference where the parameter is found. Providing specific information for the location is a great help to reviewers. The `InputTable` module is loaded via:

```
local IT = require("tex.InputTable")
```

This can be store as something other than `IT` if desired. The following functions are available via the module:

Table 2.14: InputTable Functions

Function	Description
`IT.InputTable(options)`	This function causes two tables to be generated: (1) a reference parameters table and (2) a table of parameters from other sections that are used in the current section. It also causes QuARTIC to detect the parameters that are used in equations within the section. The detection stop as soon as another another table is requested as as soon as `endInputParamDetection` is called. The `options` parameter is an optional table that can be used to specify the text that comes before each of these tables. The table fields that can be specified are 'refTableIntro' and 'calcTableIntro'. If these are not specified, a default intro is shown.
`IT.RefsTable(options)`	This function causes only the reference parameters table to be shown. As with `InputTable` this function causes QuARTIC to listen for parameters that are used in equations to populate the table. In this case only the optional field 'refTableIntro' is used.
`IT.endInputParamDetection(options)`	End detection of parameters used in equations that populate the reference tables.

The following are examples of calling the functions above:

```
@ local IT = require("tex.InputTable")
@= IT.InputTable({refTableIntro="Reference Parameters:", calcTableIntro=""})

The empty string means that no intro is included for the table that references
calculated parameters from other sections

... Add calculations here

@= IT.RefsTable()
... Add calculations here

@ IT.endInputParamDetection()
```

2.4.11.3 RELAP5 Parser Functions and RELAP5 Database Objects

QuARTIC can be used to read RELAP5 input files, which are returned as a RELAP5 database object that can be used to query for information contained in the RELAP5 model. The RELAP5

parsing library is loaded as:

```
r5 = require("relap5.RELAPparser")
```

Assuming the library is loaded to the `r5` variable, the functions for parsing a RELAP5 model are:

Table 2.15: RELAP5 Parsing Functions

Function	Description
`r5.parseRELAPfile(filename, fn)`	This takes a filename and an optional listener function of the form `fn(token, card)`. The listener is called before the card is processed, and is generally not needed. This returns a RELAP5 database object that can be used to query for RELAP5 model information.
`r5.parseRELAP5input(inputstr, fn)`	This is basically the same as the function above, but takes the file contents rather than a filename. It also returns a RELAP5 database object.

In the discusion below it is assumed that the RELAP5 database that is parsed is stored in a variable called `db`. The RELAP5 input file is organized as input lines (or cards). The RELAP5 parser reads these lines and initially converts each input line into a row of the database. These cards can be accessed as array elements. So, for example, `db[4]` gives you the 4th input line that was parsed. This is not how rows are normally accessed, but it provides some sense of how they are stored. Each input line (or database row) contains a series of input values or words. For each row of the database, a field name is associated with each input. The field name often matches the name associated with the value when it is exported by SNAP. There are some additional fields as well, such as the `kind` field, which indicates the kind of element the card is associated with. For example, this might be a component, control block, general table, etc. These various kinds of elements are numbered, and the `id` field indicates the associated number. For example, the `id` is 300 for component 300.

In order to extract useful information from the RELAP5 input file, there needs to be a way to select the information we are interested in. The database includes a `select` function for this purpose. The `select` function is used to select all rows of the database that match the select criteria. For example, the criteria might be to select all the rows (inputs) associated with component 100. Note that selecting all rows with `id` 100 may give you rows that you don't want. It may give you rows associated with component 100 and control block 100, etc. Your selection criteria could include both the `id` and the `kind` to get just the rows that you want, but this is such a common thing to request that each element type has a specific id field. For components, the id field is `cmpid` and for controls it is `cntlid`. If you select `cmpid` 100 you will only get inputs associated with component 100. These are returned as a database to allow additional queries to be performed on the rows that are returned.

The database also has a `groupby` function that allows you to create a new database where rows

are grouped together to form a single row in the new database. The grouped rows are actually a database of their own. The main database that is returned is actually a grouped database that is grouped by `kind` and `id`. Thus, there is only one row in the database for each component, control block, heat structure, general table, etc. If you select a specific component then a database is returned that contains all of the rows associated with that component. You can ten extract more specific information about the component. Note that there are some cards that are not associated with a numbered item. These cards are retained in the database, but are not grouped with any other cards.

The following are functions available on a database object:

Table 2.16: RELAP5 Parsing Functions

Function	Description
`db:carditer()`	This is used to iterate over DB rows in a `for` loop. If rows in the database are grouped into a database (such as the cards for a component), the individual cards are included in the iteration, and the grouped row is not.
`db:dbInfo()`	Return a brief summary string of the DB that indicates the number of rows among other things.
`db:dbRowInfo(recursive)`	Return a string with database row information. This shows the fields that are available in the database. Note that numbered entity rows, such as for components, are grouped in the database. To expand the fields in the grouped rows, pass in a value of `true` for the `recusive` parameter. This is a useful method for seeing what fields are available. Note that this can be called on individual rows that are returned as well. In this case, passing in `true` is probably not necessary to see the field contained in the returned item.
`db:get()`	Get the RELAP5 input associated with the selected lines. Comment lines preceeding each input row are included.
`db:getRowIdx(row)`	Get the integer index associated with a row if it is contained in the database.
`db:groupby(fields, keep)`	Given one or more field names, this groups rows together where all fields match. So for example, if the fields are 'kind' and 'id' then all rows that have the same `id`, such as 30, and the same `kind`, such as component, will be grouped together to form a single row in the database that is returned. If the keep flag is true, all rows that do not contain the specified fields are kept in the database as individual rows.
`db:iter()`	This is used to iterate over DB rows in a `for` loop. Grouped items are returned as a row item and are not expanded.

Continued on next page

Table 2.16: RELAP5 Parsing Functions (Continued)

Function	Description
`db:orderby(...)`	This can be passed one or more field names. Rows are sorted by selecting the first field in the list and sorting relative to the value. If elements are equal with respect to the first field, then the second field is selected and elements are sorted relative to that, and so on for all fields provided. Elements are sorted within the existing database, and the database is returned to allowed chained operations.
`db:select(spec, expand)`	The selection `spec` (specification) has a number of options. If it is a string, then all rows containing the field indicated by the string are returned in a new database. It can be a function of the form `fn(row)`. In this case, the function is called with each row. If the function returns a value that equates to `true` then the row is kept. If a value is returned that equates to `false` the row is dropped. The `spec` parameter can also be a table. Array argument in the table can be strings or functions and are evaluated as above. If the table contains name value pairs, then the name indicates a field in the row, and the value indicates the value the field must match to be retained. Only rows that match all elements included in the table are kept. The parameter `expand` is a boolean value. If it is specified as true will make the function return the individual rows that contain the data, not the grouped structures like components.
`db:remove(idx)`	Remove the row at the given index from the database.
`#db`	The # operator is supported and indicates the number of rows contained in the database.
`db[field]`	If the `field` is an integer, this is used to get the specified row of the database. If the field is a string, then this will loop through the rows of the database and return the value of the first row that contains the given field.

Some examples of using the RELAP5 parser and database are shown below:

```
r5 = require("relap5.RELAPparser")
db = r5.parseRELAPfile("test.i")                -- Load the database
cmps = db:select("cmpid")                       -- Select all components
for cmp in cmps:iter() do print(cmp.cmpid) end  -- Print list of components
cmp18 = cmps:select({cmpid=18})                 -- Select component 18
print(cmp18:dbRowInfo())                        -- See the fields in cmp18
L = cmp18:select("x_length"):orderby("volid")   -- Get length cards and sort
```

2.4.11.4 RELAP5 Documentation Modules

QuARTIC includes a set of modules that are useful for documenting RELAP5 components. These modules use a RELAP5 database object to generate tables that show the contents of a component. The functions contained in these modules are passed a selected component of the correct type and autogenerate tables with information about the component. The field used to select the component for the different modules are indicated in the table below:

Table 2.17: RELAP5 Documentation Modules

Module	Field to Select Entity
`relap5.ComponentDoc`	`cmpid`
`relap5.CrtlDoc`	`ctrlid` or `tripid`
`relap5.EnclosureDoc`	`enclid`
`relap5.GenTableDoc`	`gtbid`
`relap5.HeatStructDoc`	`htstid`
`relap5.MaterialDoc`	`matid`
`relap5.MultipleJunDoc`	`cmpid`
`relap5.PipeDoc`	`cmpid`
`relap5.PumpDoc`	`cmpid`
`relap5.ReactorKineticsDoc`	`kind='RKin'`
`relap5.SnglJunDoc`	`cmpid`
`relap5.TimeDepJunDoc`	`cmpid`
`relap5.TimeDepVolDoc`	`cmpid`
`relap5.ValveDoc`	`cmpid`

The modules are shown below with the functions that are available for generating tables. The first line in each table shows an example of selecting an appropriate component to document. It is up to the document author to use valid entity ids, and in the case of components, it is up to the author to call documentation functions that are compatible with the component type.

Table 2.18: Functions in `relap5.ComponentDoc`

Function	Description
`cmp = db:select({cmpid=314})`	

Continued on next page

Table 2.18: Functions in `relap5.ComponentDoc` (Continued)

Function	Description
`CmpConnectionTbl(db, cmp, filter)`	Returns a latex table with all of the components connected to a certain component. A filter function of the form `fn(row)` that is passed each row that contains a 'cmp_from' or 'cmp_to' field indicating the current component. The filter returns a boolean flag indicating if this connection should be included in the table. By default all connections to other components are listed.
`HSConnectionTbl(db, cmp, filter)`	Returns a latex table the gives all of the connected heat structures. A filter function of the form `fn(row)` that is passed rows that contain a HS row with 'left_conn_cmp' or 'right_conn_cmp' as the current component. The filter function returns a boolean flag indicating whether the heat structure row should be included in the table. By default all HS connections are included.
`CmpInfo(cmp, astable)`	Returns a paragraph describing the component with the component type, id, name, and the comment describing the component from the RELAP5 input file. The 'astable' parameter is optional, but if set to `true`, the component info is returned in a table with 'type', 'id', 'name', and 'comment' fields that provide info about the component. Note that the component does not need to be a hydraulic component - it may be a heat structure, material, control variable, etc.
`CmpIntro(db, cmp)`	Returns an introduction to the component that combines the results from the `CmpInfo`, `CmpConnectionTbl`, and the `HSConnectionTbl` functions.

Table 2.19: Functions in `relap5.CrtlDoc`

Function	Description
`ctrltbl(r5, ctrls)`	Returns a latex table containing the control type, id, and a description of the control, trips, and possibly general tables. The description contains the comment on the control and data from the control. `ctrls` is a lua table (i.e., array) that contains the following type of items: (1) a string if included becomes a title row that spans all collumns and is intended to describe the controls that follow the title in the table; (2) a positive number indicates a control variable number and causes the control to the be added as a row in the table with the control type, number and SNAP description included in the row; (3) a negative number indicates a trip and causes the trip type, number, and description to be included in the table; and (4) a component select table such as `{gtbid=5}` that is expected to be used to select a general table, but may include other types of components. Example: `{"Test controls", {gtbid = 23}, 3, 12, -100}`. This would include a title row, followed by general table 23, the control variables 3 and 12, and trip 100.
`ctrlConnectionTbl(db, filter)`	Returns a latex table that contains components that are controlled and the aspect of the component that is being controlled, as well as the control block or trip that is controlling it. A filter function of the form 'fn(row)' may be passed in that returns a boolean flag indicating whether the row should be included in the control and trip connection table or not. By default all connections are included.

Table 2.20: Functions in `relap5.EnclosureDoc`

`encl = db:select({enclid=5})`	
Function	Description
`EnclosureInfo(encl)`	Returns a latex paragraph containing the enclosure type, id, and the comment associated the enclosure.
`EnclosureHSinfo(encl)`	Returns a table with the set of heat structures that are included in the the ensclosure.
`ViewFactorsTbl(r5, encl)`	Returns a table with the view factors that are defined for the enclosure with the row and column headers being the cells included in the enclosure.

Continued on next page

Table 2.20: Functions in `relap5.EnclosureDoc` (Continued)

`encl = db:select({enclid=5})`	
`ViewFactorsList(r5, encl)`	Returns a table with the view factors that are defined for the enclosure, with each row including a single view factor. This shows the same information as the table above, but may fit on the page better.

Table 2.21: Functions in `relap5.GenTableDoc`

Function	Description
`gtb = db:select({gtbid=20})`	
`GenTblFac(gtb)`	Return a latex table with the associated trip, dependent factor, and dependent factor.
`GenTbl(gtb)`	Return a latex table with the general table data.

Table 2.22: Functions in `relap5.HeatStructDoc`

Function	Description
`hs = db:select({htstid=1024})`	
`HeatStructSummaryTbl(db)`	Summarize information about the heat structures contained in a model, such as number of cells and radial nodes, and the inner and outer radius. Note that this is not a summary for a single heat structure, but for all heat structures in the provided database. One can select a subset of the heat structures in the model and document this subset. If a single heat structure is passed in the function will fail with an error.
`HeatStructGenInfoTbl(hs)`	Another table to summarize HS general information.
`HeatStructMeshDataTbl(hs,db)`	Show radial HS properties such as material type. This requires the full database to be passed as a second parameter so that material information can be accessed.
`HeatStructInitTempTbl(hs)`	Table of initial temperatures in the HS.
`HeatStructPowSrcTbl(hs)`	Indicate the HS power source(s) and related info.
`HeatStructBndTbl(side,hs)`	Indicate the boundary conditions for one side of the heat structure. The side value must match the `bound_side` field, which supports values of 'left' or 'right'.

Continued on next page

Table 2.22: Functions in `relap5.HeatStructDoc` (Continued)

Function	Description
`HeatStructAddBndTbl(side,hs)`	Additional boundary condition info table. Again the `side` should be 'left' or 'right'.

Table 2.23: Functions in `relap5.MaterialDoc`

Function	Description
`mat = db:select({matid=4})`	
`MaterialInfo(mat)`	Get a paragraph desribing the material that includes the material id, name, description of the material type, and the material comment included in the input file.
`PropertiesTbl(mat)`	Show the material properties tables associated with this material.

Table 2.24: Functions in `relap5.MultipleJunDoc`

Function	Description
`cmp = db:select({cmpid=15})`	
`MultJunInfoTbl(cmp)`	Gives initial conditions and general information such as the discharge coefficient and the initial liquid velocity.
`MultJunAddInfoTbl(cmp)`	Another table that gives other general information such as the junction hydraulic diameter and the slope.
`MultJunFlags(cmp)`	Junction flag info table.
`MultJunLossTbl(cmp)`	Gives the flow loss equations for the different junctions.
`MultJunConnectionTbl(cmp, db)`	Gives the different components that are connected multiple junction. It tells which ones are inlets and outlets, and it gives their area.

Table 2.25: Functions in `relap5.PipeDoc`

Function	Description
`cmp = db:select({cmpid=50})`	
`PipeVolGeomTbl(cmp)`	Table of geometric properties such as cell lengths, areas, volumes, elevation change, and angles.
`PipeVolFricTbl(cmp)`	Table of wall roughness and hydraulic diameter.

Continued on next page

Table 2.25: Functions in `relap5.PipeDoc` (Continued)

Function	Description
`PipeVolIC(cmp)`	Volume initial conditions info.
`PipeVolFlags(cmp, dir)`	Volume flag information table for direction 'x', 'y', or 'z'. Default `dir` is 'x' (axial).
`PipeJunFlowLoss(cmp)`	Junction K loss factors.
`PipeJunIC(cmp)`	Junction initial conditions table.
`PipeJunFlags(cmp)`	Junction flags table.

Table 2.26: Functions in `relap5.PumpDoc`

Function	Descrition
`cmp = db:select({cmpid=42})`	
`PumpCurveTbl(cmp, regimes, degraded)`	Table of the given regime values. These regimes can be from the head and torque curves or the degraded head and torue curves. The 'regimes' parameter is as list of one or more of the following pump regimes: (e.g., {'HVN','HVD',...}). A column will be added for each regime in the table. The 'degraded' parameter is optional. By default it is `false`. To get degraded regimes, pass in `true`.
`PumpMultiplierTbl(cmp, dtype)`	The multiplier table for either the head multipliers, or the torque multipliers.
`MotorTorqueCurve(cmp)`	Gives the motor torque table of the pump.
`PumpVelocity(cmp)`	Gives the pump velocity information.
`PumpFlowLoss(cmp, inOut)`	Gives the flow loss equation for either the inlet or outlet in a table format.
`PumpDescTbl(cmp)`	Gives a table of the initial conditions of the pump.
`PumpGeom(cmp)`	Gives the pump geometry information such as area, length, and volume.
`PumpVolIC(cmp)`	Generate a table with pump volume initial conditions.
`PumpJunIC(cmp)`	Generate a table with pump junction initial conditions.
`PumpVolFlags(cmp, dir)`	Volume flag information table for direction 'x', 'y', or 'z'. Default `dir` is 'x' (axial).

Table 2.27: Functions in `relap5.ReactorKineticsDoc`

Function	Descrition
`RK = db:select({rkinid=0})`	
`ReactorPoerInfoTbl(RK)`	Returns a latex table with the power
`ReactorFissionProductTbl(RK)`	Returns a latex table with the fractions of the power that various materials give.

Table 2.28: Functions in `relap5.SnglJunDoc`

Function	Description
`cmp = db:select({cmpid=20})`	
`SnglJunConnectionTbl(cmp,db)`	Connections to other components.
`SnglJunFlowLoss(cmp)`	K loss information.
`SnglJunIC(cmp)`	Initial conditions table.
`SnglJunFlags(cmp)`	Junction flags info.

Table 2.29: Functions in `relap5.SnglVolDoc`

Function	Description
`cmp = db:select({cmpid=20})`	
`SnglVolGeomTbl(cmp)`	Table with geometry of the volume.
`SnglVolFricTbl(cmp)`	Hydraulic diameter and wall roughness table.
`SnglVolIC(cmp)`	Initial conditions table.
`SnglVolFlags(cmp, dir)`	Volume flags info for 'x', 'y', or 'z' direction. Default `dir` is 'x' (axial).

Table 2.30: Functions in `relap5.TimeDepJunDoc`

Function	Description
`cmp = db:select({cmpid=13})`	
`TimeDepJunConnections(cmp, db)`	Table of the connections to the time dependent junction.
`TimeDepJunFlowTbl(cmp)`	Flow control table for the Time Dependent Junction.

Table 2.31: Functions in `relap5.TimeDepVolDoc`

Function	Description
`cmp = db:select({cmpid=202})`	
`TimeDepVolGeomTbl(cmp)`	Show geometric information.
`TimeDepVolFlags(cmp)`	Show volume flags.
`TimeDepVolBCTbl(cmp)`	Show the time dependent volume boundary conditions control table. The controls are described before the table.

Table 2.32: Functions in `relap5.ValveDoc`

Function	Description
`cmp = db:select({cmpid=202})`	
`ValveConnectionTbl(cmp,db)`	Show connections to other components.
`ValveType(cmp)`	Valve type information
`ValveFlags(cmp)`	Table of junction flags.
`ValveFlowLoss(cmp)`	K loss factors.
`ValveIC(cmp)`	Junction initial condition info.
`ValveCntrlParams(cmp)`	Valve control parameters.
`ValveCsubv(cmp)`	Valve flow coefficients.

Below is an example of adding Valve tables to a document. The process for including tables for other components follows the same pattern.

```
@ r5prs = require("relap5.RELAPparser")      -- Include the parser library
@ db    = r5prs.parseRELAPfile("test.i")     -- Load the database
@ PD    = require("relap5.PipeDoc")          -- Include the pipe doc library
...
\textsc{\large Pipe 15} \nopagebreak
@ pipe15 = db:select({cmpid=15})

The pipe geometric parameters are calculated as ...

@[= PD.PipeVolGeomTbl(pipe15) ]@

The flow K loss factors are calculated to be ...

@[= PD.PipeJunFlowLoss(pipe15) ]@
```

PIPE 15

The pipe geometric parameters are calculated as ...

Range	Length (m)	Area (m^2)	Volume (m^3)	Elevation Change (m)	Vert Angle (deg)	Azim Angle (deg)
1 - 12	0.1476	0.01867	0.002756[a]	-0.1476	-90.0	0.0
13	0.1524	0.0252	0.00384[a]	-0.1524	-90.0	0.0
14	0.1365	0.03228	0.004406[a]	-0.1365	-90.0	0.0

The flow K loss factors are calculated to be ...

Range	Fwd Loss	Rev Loss
1 - 11	0.0	0.0
12	0.07345	0.053446
13	0.0	0.0

2.4.11.5 ParamTable Module

The `ParamTable` module is used to compare parameters that are calculated in the QuARTIC document with RELAP5 model fields to verify that the values match. It also may be used to generate a table of the parameters that are used to list the parameters that are used to set fields in the RELAP5 model.

Table 2.33: Functions in `relap5.ParamTable`

Function/Variable	Description
`tolerance`	This module variable indicates the relative tolerance in the difference between the QuARTIC parameter and the associated RELAP5 model field value. The default tolerance is 0.0001. Note that this is a relative tolerance meaning that it is compared to the difference between the parameter and the field value divided by the average of the two values. A difference that exceeds the tolerance is marked as a mismatch.

Continued on next page

Table 2.33: Functions in `relap5.ParamTable` (Continued)

Function/Variable	Description
`cmpPrm2Model(prm,cmp,fld,cells,tol)`	Compare the given QuARTIC parameter (`prm`) to the RELAP5 model field (`fld`) for the selected component (`cmp`) over the given volume, junction, or node range (`cells`). If the value is global to the selected component, then this parameter is not included. An optional tolerance (`tol`) can be specified if the default is not adequate. This function just registers a parameter for comparison. The actual comparison occurs when `modelParamReport` is called.
`paramTable(cmp,fld,rows)`	The `paramTable` function registers parameters for comparison to model fields and also generates a table showing the parameters. The `cmp` value is a selected component from the RELAP5 parameter database. The field (`fld`) is a string indicating the a RELAP5 field name to compare the parameter against. The field may be included muliple rows of the component database. Sometime a field name is insufficient to select the rows for comparison an a `select` specification table is used. This will be described in more detail in this section. The `rows` value is a table containing one or more rows with each row containing a parameter to compare the model field against and information about which volume, junction, or node has the field to compare against (when applicable). The structure of this table and its rows is shown later in the section.

Continued on next page

Table 2.33: Functions in `relap5.ParamTable` (Continued)

Function/Variable	Description
`modelParamReport(opts)`	This function causes a model parameters report to be generated to a csv file. It should be executed after all parameters parameters have been calculated. A good practice is to include this as a delayed function so that it is processed after the document is completed. The delayed call to this function may be placed at the start of the document for convenience if desired. The optional (`opts`) parameter can be a string that indicates the output csv file name ('ModelParamReport.csv' by default), a real value that overrides the default tolerance, or a boolean that if `true` indicates that only mismatch values should be shown in the table. This can also be a table with the fields `filename`, `tol`, or `onlyMismatch` to set any or all of the parameters above.

The intent of the `paramTable` function is to register a set of parameters for comparison against model fields, and at the same time, generate a table of parameter values that provides a reference for where in the document the parameters are found. It is expected that this table will be located in a section where tables of the RELAP5 input parameters from Section 2.4.11.4 are shown. The parameter table is intended to show the mapping between parameters calculated in the QuARTIC document and fields in the RELAP5 model - typically fields from a specific component.

Prior to calling `paramTable(cmp, fld, rows)`, a component is selected from a RELAP5 parameter database object (see the tables in Section 2.4.11.4 for examples of selecting various components). A RELAP5 model field name to compare against must then be specified. A common value might be something like `'x_length'`. Note that there are likely to be multiple rows in a component that specify axial length and the `rows` parameter, discussed shortly, will allow us to specify different parameters to compare against different rows that include the `'x_length'` field. The select rows are now ready for comparison against the set of parameters defined in the `rows` value that is passed to `paramTable`.

The `rows` parameter is a table containing one or more rows, with each row indicating a parameter to compare against and a specification for which of the selected rows (volumes/junctions/nodes) to compare the parameter against. Each row of the table specifies a parameter to compare against, and possibly an integer indicating the number of volumes/junctions/nodes that match the parameter value. If the integer value is not included, this implies one volume/junction/node matches the parameter. Below is a simple example that is used to generate a table of calculated lengths and compare these against the `'x_length'` field in the model.

```
@[
relap5 = require("relap5.RELAPparser")
db = relap5.parseRELAPfile("input/Pipe.i")
PT = require("relap5.ParamTable")
]@
@? PT.modelParamReport()
...
@ pipe = db:select({cmpid = 3})
@= PT.paramTable(pipe, "x_length", {{L_1}, {L_2, 3}, {L_3}})
```

The first row is `{L_1}`, indicating that parameter `L_1` is compared against the axial length of volume 1. The second row is `{L_2, 3}` indicating the axial length of volumes 2, 3, and 4 should match `L_2`. The third row is `{L_3}` indicating the volume 5 should match the length `L_3`. A table is generated from this code as shown below.

...

Parameter	Value	Reference	Loc / Note
L_1	5 m	Equation (2.4.1) pg. 17	Cell 1
L_2	6 m	Equation (2.4.2) pg. 17	Cells 2-4
L_3	3 m	Equation (2.4.3) pg. 17	Cell 5

In order for parameters to be compared against model fields, it is necessary to call the `modelParamReport` function. Note that in the example above, this function is called with the delay modifier [?] (see section section 2.4.2.1.1)), so that this function is run after the document is completed and all parameters and fields have been registered. As noted in the table above, the `modelParamReport` function writes a csv file that compares the parameters included in calls to `paramTable` with the associated value in the RELAP5 input file.

The `modelParamReport` calculates a relative difference between the QuARTIC parameter and the value of the specified field in the RELAP5 model. The relative difference is calculated by $|2 \cdot (P\_calc - P\_model)/(P\_calc + P\_model)|$). If the relative tolerance is larger than the module default tolerance, or a specified tolerance, then the field is marked as a mismatch in the csv file in the last column. This helps provide some check on the consistency of calculated parameters vs. the actual model. The resulting table has contents like the following:

The file that is generated has the following form.

```
Param,Value,Units,Card,Cmp ID,Cells,R5 Field,vol-jun-node,R5 Value
L_1, 5, m, 30301, pipe 3, cell 1, x_length, 1, 5.0,
L_2, 6, m, 30302, pipe 3, cell 2-4, x_length, 3, 6.0,
L_2, 6, m, 30303, pipe 3, cell 2-4, x_length, 4, 6.1, mismatch
L_3, 3, m, 30304, pipe 3, cell 5, x_length, 5, 3.01, mismatch
```

Below is an example that adjusts the file that is output and the tolerance that is used for comparisons.

```
@? PT.modelParamReport({filename = "ModelParamComparison.csv", tolerance =
    0.00034})
```

In some cases, specifying a field name alone for the `fld` parameter of the `paramTable` function is insufficient to select the rows with the set of fields to compare against. In this case a parameter database `select` specification table is defined that allows the rows containing the correct fields to be selected for comparison. This must contain the field for comparison as the first entry in the selection table. For example, in heat structures, both the left and right boundary condition may contain a `'factor'` field, which can represent a heat structure surface area or a cylinder length, among other things. Generally, when comparing factors to parameters calculated in the document, you need to compare against either the right or left boundary condition. This is done by setting the `fld` value to a table such as `{'factor', bound_side='left'}`. This causes rows to be selected that contain the 'factor' field, where the `'bound_side'` property is `'left'`. Alternatively, the `'right'` boundary could be selected. Once a set of rows are selected, they are ordered by node, volume id, or junction id if any of these are relevant to the selected rows.

```
@ hs = db:select({htstid = 104})
@= PT.paramTable(hs, {'factor', bound_side='left'}, {{L_1}, {L_2, 3}, {L_3}})
```

The individual rows in the `rows` entry for `paramTable` provide significant capability to control the parameter to compare against. Below are fields that can be included with a row to control how comparisons are managed:

- `cell` can be specified to indicate the end of a range of cells to compare the parameter against. For example, `{L_1}, {L_2, cell=6}, ...` the first index against `L_1` and values 2 through 6 to `L_2`. If the value is not associated with a volume/junction/node, then use `cell="none"`.
- `initcell` resets the initial cell in a range. This can be done on the first row to start with a different volume/junction/node index than one, or it may be use on some later row to skip cells or start the range over (e.g., `{{L_1, initcell=10}, ...}` starts at cell 10).
- `decreasing=true` can be included to indicate that nodes should be compared in decreasing order. This would typically be specified in concert with `initcell` (e.g., `{{L_1, 3, initcell=10, decreasing=true}, ...}` compares `{L_1}` against cells 10, 9, and 8).
- `field` is used to change the field to compare against. Typically this will be accompanied by `initcell` (e.g., `{A_1, initcell=1, field="x_area"}`)
- `tol` is used to set the tolerance used for the current row and rows that follow.
- `note` is used to include a note with the parameter in the parameter table.

The following example uses more of these features:

$$A_{1x3} = 0.5 \text{ m}^2 \tag{2.4.27}$$

$$A_4 = 0.3 \text{ m}^2 \tag{2.4.28}$$

$$A_{5x6} = 0.2 \text{ m}^2 \tag{2.4.29}$$

$$Th_{1x2} = 8e{-}6 \text{ m} \tag{2.4.30}$$

$$Th_3 = 9e{-}6 \text{ m} \tag{2.4.31}$$

$$Th_4 = 1e{-}5 \text{ m} \tag{2.4.32}$$

```
@ hs = db:select({htstid = 11})
@[= PT.paramTable(hs, {'factor', bound_side='left'}, {{A_1x3, cell=3,
    note="Areas"},
                                                        {A_5x6, 2, initcell=5},
                                                        {Th_4, field="thickness",
    initcell=4, decreasing=true, note="Thickness", tol=1e-4},
                                                        {Th_3},
                                                        {Th_1x2, 2, note="Last
    value"}}) ]@
```

Parameter	Value	Reference	Loc / Note
A_{1x3}	0.5 m^2	Equation (2.4.27) pg. 60	Cells 1-3 Areas
A_{5x6}	0.2 m^2	Equation (2.4.29) pg. 60	Cells 5-6
Th_4	$1e{-}5$ m	Equation (2.4.32) pg. 61	4 Thickness
Th_3	$9e{-}6$ m	Equation (2.4.31) pg. 61	3
Th_{1x2}	$8e{-}6$ m	Equation (2.4.30) pg. 61	s 1-2 Last value

3 Debugging QuARTIC Template Errors

In order to effectively debug QuARTIC errors, it is useful to understand how QuARTIC generates a document. When QuARTIC is run to process a template file, a few steps occur:

1. QuARTIC processes the template file and converts it to a Lua script file. The name of the template file, and the location of the template file folder are specified via the command line. The Lua script file that is generated is then saved the the folder containing the QuARTIC scripts in a folder named 'tmp'.

2. QuARTIC then runs the Lua script to generate a lateX file that is ready for processing. This is placed in the output folder specified on the command line, and has the same name as the original template file.

3. At this point QuARTIC exits, and latex is called to process the file that was generated.

Errors are reported during stages 2 and 3. Template syntax errors, which can include things like using invalid Lua code, or fogetting to close a script block, are not generally noticed by the QuARTIC compiler, but will typically result in as error that is reported when the QuARTIC tries to run the Lua script file, or alternatively when latex is executed to generate the PDF document.

Errors that are reported by QuARTIC occur before latex is called, and latex will not be run if QuARTIC reports an error. QuARTIC will indicate a line number where the error occurred. The line number is a line number relative to the Lua script file that is generated by QuARTIC.

Latex errors are reported by the latex program that is used to process the latex file. Latex also reports line numbers for errors. In this case the line numbers are associated with the final latex document that is produced, and NOT the template file.

When fixing errors reported by QuARTIC, it is useful to have the template file and the Lua script file open. When fixing errors reported by latex, it is useful to have both the template file and the final generated latex file open for examination. Since the line numbers reported in the error are relative to the Lua script file or the final generated latex file, these files are useful to determining where in the template file to look to fix the problem. NOTE that errors MUST be fixed in the QuARTIC template file. Any changes to the Lua script file, or the generated latex file will be overwritten when QuARTIC is executed!

In general, when looking for errors reported in the Lua script file or in the latex file, it is useful to look at the line reported as well as at the few lines preceeding the error line. Often the error actually occurs on the line before the line reported. Note that both the Lua script file and the final latex contain text from the template file. The text surrounding the error is useful for determining the location of the error in the QuARTIC template file, so that the error can be located and fixed

in the template file.

Some knowledge of Lua syntax and latex syntax is useful when identifying errors. However, it is often sufficient to look for differences from other similar constructs in the script or generated latex document. Note that it is sometimes easier to spot errors in the QuARTIC template file. Other times, it is easier to spot errors in the Lua script file or the generated latex file.

3.1 Example of Fixing Error Reported by Lua Script

In the example below, a typo occurred where the function 'References' got replaced with the a call to a nonexistent function 's'. The example template code is shown below:

```
%===============================================================================
\chapter{Forces Description}
%===============================================================================

%===============================================================================
@[=? s() ]@
%===============================================================================
```

When this is executed QuARTIC reports the following error (note that latex does not run because QuARTIC failed to create the generated latex document):

```
----------------------
Generating Latex Files
----------------------
Writing '../../CalcNotebook/latex/HydroLoadsCalcNotebook.tex'
./tmp/HydroLoadsCalcNotebook.lua:1637: attempt to call global 's' (a table value)
```

The error is reporting that 's' is being 'called' as if it is a function, but it is not a function. Since an invalid function call is being made and QuARTIC fails. The contents of 'HydroLoadsCalcNotebook.lua' is shown with line numbers.

```
1628  _ENV:_write_([[
1629
    %=============================================================================
1630  \chapter{Forces Description}
1631
    %=============================================================================
1632
1633
1634
    %=============================================================================
1635  ]])
1636  _ENV:_write_(function() return
1637  s()  end)
```

On line 1637 we see ‘s()‘, which is the Lua syntax for calling a function named ‘s’. The lua code file has similarity to the template file, with some extra text added. This file generally gives enough context for you to do a text search in the QuARTIC template file to find where the error is occurring. In this case, there is no syntax error to fix - the function that is being called simply doesn’t exist. At this point, you may recognize that something changed that you did not intend. Or maybe the function name is spelled wrong. If you don’t recognize the change, assuming a source control tool such as SVN is being used to track changes to the file (which is highly recommended), it is useful to look and see if an unexpected change occurred in this location in the file. Fixing the error may be a simple matter of reverting the change to the file.

3.2 Using the Builtin Lua Debugger

Sometimes it may be difficult to determine from an error message what the problem is. QuARTIC has a builtin Lua debugger module that can be used to step through generation of the document line by line. To use this, include the following in the template file above the place where you would like to start stepping through the document:

```
@ dbg()
```

The debug session will look something like the text below:

```
@Clua\CalcNotebook.lua:77 in 'template'
72
73    \usepackage{zref-abspage}[2010/03/26]
74
75    ]])
76     dbg()
77 => _ENV:_write_([[
78
79    \makeatletter
80    % Counter 'topic@label' for automatic generation of label names
81    \newcounter{topic@label}
82    \renewcommand*{\thetopic@label}{topic@\the\value{topic@label}}
debugger.lua (h for help)>
```

This shows a few lines from the lua file that you are debugging. The file is indicated along with line numbers. The arrow (=>) points to the line number that is about to be run. A debug command prompt is shown. Typing 'h' followed by enter prints the list of debugging commands. There are commands for stepping through the code line by line or stepping into functions that are called. Code is not available for the internal QuARTIC functions, but will be available for the compiled template files and any other custom functions that are generated for the document.

If you are using the debugger from a Lua code file, then the debugger module has to be imported and then called as shown below:

```
local dbg = require("debugger")
dbg()
```

3.3 Example of Fixing Error Reported by Latex

After running QuARTIC, latex is used to process the file. The version of latex that QuARTIC has been tested with is lualatex. Latex prints a lot of information when processing files. Many things might look like errors, but typically most of the information is unimportant. It is usually just the final few lines that are important. Below is an example of an error reported by lualatex:

```
Underfull \vbox (badness 3525) has occurred while \output is active [9]
Underfull \vbox (badness 1496) has occurred while \output is active
    [10<./Figures/FigA6CrosbyLoopSeal.jpg>]
./HydroLoadsCalcNotebook.tex:500: Missing $ inserted.
<inserted text>
$
l.500

?
```

The '?' at the end is a latex prompt asking what you would like to do. Often you don't need to do

anything other than close the window after you are done locating the error information. A couple lines above the '?' is 'l.500'. This indicates that the error was detected at line 500. Often the error actually occurs on the line before this. A couple of lines above this is a message that indicates the name of the file (HydroLocaCalcNotebook.tex) and the line number (500) where the error occurred, followed by a description of the error. The error indicates that a '$' is missing. In latex, $ are used around inline math expressions. And this error may indicate that there is an unexpected $ added to the latex code.

```
496  \begin{align}
497  DX_{5.c4} = \frac{R_{SW663.and.665}}{2} = \frac{0.2286~\mathrm{m}}{2} =
    0.1143~\mathrm{m} \label{eq:DX_5_c4}
498  \end{align}
499  $$@
500
```

Here we see an extra '$$@' symbol. These are the symbols used to close a symbolic equation is QuARTIC (see Section 2.4.2.2), so it is probable that there was an error in the syntax used to add an equation. To figure out where to look in the template document, you can look for some surrounding text. Alternatively, you can search for the variable associated with `DX_{5.c4}`. Note that QuARTIC converts variable names used in the template file to names that are valid in latex. After the first underscore, it replaces underscores with dots. All QuARTIC variable names must have valid Lua variable names. Letters, numbers, and underscores are allowed, but no other symbols. Curly braces are used a lot in latex, so the first thing to do to recover the variable name is eliminate curly braces (i.e., `DX_5.c4`). As noted above, the dot is replaced with an underscore resulting in the QuARTIC variable name `DX_5_c4`. This is the variable name to search for in the template file. Search for this finds the following:

```
402  @$$>1
403  DX_5_c4 =  R_SW663_and_665/2 $$@
404  $$@
```

Note that there is an end of equation tag at the end of line 403 and there is also an end of equation tag on line 404. This is a scripting error. After removing the end of equation tag from line 403, the equation is formatted correctly.