TU DELFT

PQ9 Code Generator Manual

Subsystem software from dynamic templates

Created by:
Erik Mekkes

January 19, 2019

Contents

Chapter 1

About the Code Generator

The PQ9 Code Generator is a template system that is specifically written for generating PQ9 Subsystem software. The goal is to provide a program that makes the subsystem software accessible to those without in-depth understanding of either the PQ9 software structure or the subsystem specific hardware code by generating the code from templates and parameters.

The Code Generator comes with a custom template format, for which a number of commands and keywords have been defined to make the templates very versatile. The templates and Code Generator can be used for any application that benefits from templates and parameters. The generator is originally intended to use templates that produce code in the C programming language for use within the PQ9 system, but the Code Generator doesn't explicitly require this. Technically any text or code language should be accepted by the the code generator as a valid template, as long as the template contents don't conflict with the pre-defined template commands and keywords (See chapter 5).

The intended use is to provide a standard code structure that is shared among hardware within the PQ9 system. This structure can be represented as a set of base templates for a PQ9 subsystem. Subsystem experts can then be asked to fill in the subsystem specific code sections of these templates for their subsystems. An extra burden is placed on the subsystem experts by further asking them to define their code in terms of subsystem parameters, for as much as such a request is possible. Thus producing a subsystem specific set of templates and parameters, which when run through the Code Generator produce fully functional subsystem software for the specified parameters.

This could be used to work towards a standard for PQ9 software that is consistent across subsystems. It would also allow for simple modification of a subsystem's behaviour by changing subsystem parameters, without having to consider underlying code. Similarly, the base templates should make implementing hardware specific code sections easier to manage by not having to consider the overhead of the PQ9 software structure.

Chapter 2
Quick Start Guide

2.1 Where to get the Code Generator

For a direct download of the latest version of the PQ9 Code Generator, including this manual, the source code and example templates, head over to the latest release on the GitHub repository:

https://github.com/ErikMekkes/PQ9_bus_software/releases

2.2 Running the code generator

Update the settings.json file :

- Change the desired name for the subsystem.
- Update the list of desired sub-directories.
- Update the list of desired files to generate, for each file :
  - provide the output filename
  - provide the base template name if it's different from the output filename
  - provide the set of parameters used within the output file (as list or as csv file)

Check / fill in your specified .csv files (or update the params.csv file when using list).

Check / fill in your specified base templates, make sure to use to /cgen_template extension.

Start the code Generator by running CodeGenerator.jar

2.3 Modifying the source code

The entire code generator project is open source, you are free to make your own copy, clone or fork of the original source files and free to customize it to your needs.

The project is set up with maven, for which the configuration has been included. It should be possible to directly import the source code as a maven project and modify / build / run it in most IDE's. Maven has been configured to create an executable jar file in ./target/CodeGenerator directory, along with all required external files such as templates and settings.json during the build cycle. If you wish to include additional external files, please check out the resources section and comments in the maven pom.xml.

Have a contribution that you believe should be added to the original? Make a fork of the original github repository, apply your changes and create a pull request to the code generator repository.

2.4 Sharing templates

Currently there is no service provided by the maintainers to share templates. Feel free to set one up if there is a demand.

Chapter 3

Program Settings

The program comes in the format of an executable jar file, which can be run directly from the desktop or command line on most operating systems. The program looks for a settings.json file in the same directory for instructions on what to generate. If a settings file isn't present, the program will print a warning and exit. The settings file uses the json format, an example settings file and an explanation of the entries is shown below in Listing 3.1.

Listing 3.1: settings.json

```
{
  "subsystem_name" : "ADCS",
  "subdirectories" : ["HAL"],
  "parameters" : "params.csv",
  "auto_increment_start_id" : 44,
  "continue_indentation" : true,
  "overwrite_existing_files" : true,
  "clear_existing_directories" : false,
  "logging" : true,
  "logfile" : "CodeGenerator.log",
  "files_to_generate" : [
    {
      "filename" : "parameters.c",
      "base_template":"parameters.c.cgen_template"
    },
    {
      "filename" : "parameters.h"
    },
    {
        "filename":"../satellite.h",
        "base_template":"satellite.h.cgen_template"
      }
  ]
}
```

subsystem_name Default : generated_subsystem. The name of the subsystem to be generated, used for the name of the output folder and available in templates with the s#name keyword (5.1.1).

subdirectories Default : none. The desired sub-directories to create in the main output folder as a list of directory names, format : ["directory_name_1", "directory_name_2", ...].

Use an empty list if no sub-directories are required : []

parameters Default : params.csv. A comma seperated values (.csv) file describing the available parameters for this subsystem. If a different parameters file should be used it can be specified here. See Section 4.1 for an example params.csv file.

auto_increment_start_id Default : disabled. It is possible to leave the parameter ids in the parameters file unspecified and instead specify a start value. If this option is specified the parameters in the .csv file will be assigned incrementing ids from this starting value corresponding to their order in the .csv file.

Leave out this setting or set it's value to -1 to disable.

continue_indentation Default : enabled. Specifies whether the program should automatically continue with the same level of indentation when inserting sub-templates.

overwrite_existing_files Default : enabled. Specifies what should happen if a file to be created by the code generator already exists. If enabled existing files will be overwritten.

clear_existing_directories Default : disabled. Instructs the program to clear the contents from the specified subsystem output directory under subsystem_name if the directory already exists.

WARNING : this deletes all files within the specified directory, this is not reversible!

logging Default : enabled. Specifies whether a logfile should be created of the program's progress and warnings / errors.

logfile Default : CodeGenerator.log. Specifies where the logfile should be created

files_to_generate Default : none. Specifies the list ([... , ...]) of files to generate. Each file should be listed as a JSONObjects ({ ... , ...}), and should at least specify a filename.

The available attributes for file JSONObjects are :

- filename : The name for the output file, can be a relative path such as "../filename" or "HAL/-filename".
- base_template : The base template for the file, should have the .cgen_template extension. If not specified the generator looks for 'filename.cgen_template'.
- parameters : Allows a subset of the parameter csv file to be specified for this file. Useful when frequently making use of the 'all' keyword (See Section 5.3). Format is a list of parameter names : [name_1, name_2, ...].

Chapter 4

Specifying parameters

One of the main objectives of the code generator is to make it easy to modify a sub-system's code by only having to change the parameters that were defined for it. In order to do so the desired code is generated from a set of templates containing the desired subsystem code in terms of subsystem parameters, and an external comma separated values file that specifies these parameters. This chapter briefly covers the accepted format for the .csv file using an example antenna deployment board's (ADB) parameters.

4.1 Example params.csv file for an ADB subsystem

```
1,testing_4,uint32_t,0xCAFE
2,SBSYS_sensor_loop,uint32_t,60000
3,adb_sensor_status,uint8_t,0xDEAD
4,adb_int_temp,uint16_t,12
5,adb_deb,uint16_t, 4000
6,SBSYS_reset_clr_int_wdg,uint32_t,8000
7,SBSYS_reset_cmd_int_wdg,uint16_t,600
```

Each row represents a parameter and contains the following entries :

- id : the number id of the parameter (should be unique within the PQ9 system)
- name : the name of the parameter (should be unique within the PQ9 system)
- datatype : the data type of the parameter (c programming language)
- default value : the default value for the parameter (matching datatype)

The number can be left unspecified as -1 if auto-incrementing is enabled in the program settings. The numbers will be automatically assigned starting from the specified starting number. The program will print out warnings if auto-incrementing is enabled and ids aren't left unspecified as -1 to prevent unintended ids.

Chapter 5

Working with templates

The program builds the desired subsystem code from a specified set of templates and parameters as described in the previous chapters. The templates should form a code framework that generates the actual desired code based on the parameters. This means that creating subsystem templates will require expertise with the PQ9 software architecture and the subsystem hardware.

Though intended for the PQ9 architecture, the templates can contain anything, as long as they are files with the .cgen_template extension. This chapter explains the available options within templates, which come in two variants: keywords and commands.

5.1 Template Keywords

Keywords are pre-defined words that are recognized by the program. This section lists the keywords that can be used within templates. Just like keywords from programming languages these keywords can not be used for anything other than their pre-defined purpose, the code generator cannot distinguish between other intended uses, so will always interpret them with these pre-defined meanings.

None of the template keywords are valid c-code, they will not compile, and the characters used in them generally aren't used in such a way to produce valid code for programming languages. The template keywords have been chosen in such a way to make it extremely unlikely for them to be mistakenly used for anything other than their intended purpose.

5.1.1 Subsystem Name

The s#name keyword can be used to substitute the subsystem name in templates. The value is taken from the subsystem name specified in the settings file.

Example, if the subsystem name was specified as `ADCS`:

```
s#name_sensor_value = 0;
```

will generate as output:

```
ADCS_sensor_value = 0;
```

5.1.2 Parameter Properties

These keywords are pre-defined to represent parameter properties, they are replaced with the corresponding values of specified parameter(s) during the code generation process (see 5.3).

- p#name : The name of the parameter
- p#id : The integer id enum value used to indentify the parameter.
- p#enumName : The name used to identify the enum value of the parameter. Equal to the string 'p#name_param_id'.
- p#dataType : The data type of the parameter.
- p#defaultValue : The default value of the parameter.
- p#hexId : hexadecimal representation of the id. (for XML templates)
- p#dType : alternative data type name format (short, long, ...). (for XML templates)

5.2 Generic Commands

Commands are pre-defined line formats that instruct the program to do something. They are identified by the surrounding dollar $ signs.

5.2.1 Variables

The `$var$` command is used to define a template variable for the current template, the format is as follows:

```
$var$ \{variable_name\} variable_value
```

The effect is very straightforward, every instance of variable_name in the template is replaced with variable_value. The replacement proceeds from the beginning of the template to the end, for example, replacing ”aa” with ”b” in the string ”aaa” will result in ”ba” rather than ”ab”.

Anything before the command identifier is ignored, so type and amount of leading whitespace can be freely chosen.

Anything between the escaped braces following the `$var$` command is taken as the variable name, so anything except escaped braces is a valid variable name. For example `foo_var` is a valid variable name, but so is the string `this gets replaced by the value` or just the number `5`.

The variable name must be followed by a single space character, anything following this single space character is accepted as a value for the variable, as long as it is on the same line. For example the word `bar` or the number `4` is an accepted value, but so is `*((uint32_t*)value)` or `I actually want to replace foo_var with this entire sentence`.

Variables only affect the remainder of the template after their definition. For a full example of the effect and possible uses, see example A.1. Additionally, template commands aren’t effected by variables.

Import variables from file The `$vars$` command is available in addition to the `$var$` command. It allows a template file containing variable name and value pairs to be specified. The variables are then loaded for the current template:

```
$vars$ variables.cgen_template
```

It is assumed that the content of the template file consists only of variable name and value pairs:

Listing 5.1: variables.cgen_template

```
\{variable_name_1\} variable_value_1
\{variable_name_2\} variable_value_2
\{variable_name_3\} variable_value_3
...
```

5.2.2 Subtemplates

The `$template$` command is used to include another template in the current template, the format is as follows:

```
$template$ template_filename
```

The specified template file will be processed as a sub-template, and this command line will be replaced with the output from that sub-template. The existing variables and parameters specified for the current template are passed along to the sub-template. New template variables defined in the sub-template will only be available locally within the scope of that sub-template.

It is possible to redefine existing variables in a sub-template with a different value, the re-defined value will only apply within the scope of the sub-template. The code generator displays a warning when values from parent templates are redefined in sub-templates to prevent accidental re-use of variable names.

Template processing is performed recursively, sub-templates may again contain commands and thus may define further sub-templates and so on.

5.3 Parameter Processing Commands

These commands can be used to process a template section for a specified set of parameters. Along with the command itself a list of parameters is specified, the command is executed once for each parameter in that list.

The accepted format for this parameter list is :

```
[param_name_1|param_name_2|param_name_3|...]
```

The parameter names used in this list should be a subset of the available parameters specified for this template in the settings. The program will display a warning on execution if a parameter was not recognized.

Additionally, the keyword `all` is supported as a valid list entry. When used the command will be executed for all parameters that were specified for this template in the settings.

5.3.1 Templates

The `$p-template$` command can be used to include a sub-template once for each parameter in the specified parameter list. For each evaluation of the command, the corresponding parameter's values are filled in for parameter keywords (see 5.1.2). The format is as follows:

```
$p-template$ list_of_parameters template_filename
```

Template processing is performed recursively, sub-templates may again contain commands and thus may define further sub-templates and so on. This includes the ability to specify new parameter commands while the parent template is being processed for a certain parameter.

5.3.2 Blocks

The `$p-block$` command can be used to include a block of code once for each parameter in the specified parameter list. It is similar to `$p-template$`, but without the need for other template files. For each evaluation of the command, the corresponding parameter's values are filled in for parameter keywords (see 5.1.2). Anything between the escaped braces is considered part of the block of code to include. It is assumed that the escaped closing brace is on it's own line, use `$p-line$` (5.3.3) instead for shorter inclusions.

```
$p–block$ list_of_parameters \{
   code goes here;
\}
```

There are no format restrictions for the code_line input, any remaining characters on the current line are included, the only requirement is that it is separated from list_of_parameters by a single space character.

5.3.3 Lines

The `$p–line$` command can be used to include a line once for each parameter in the specified parameter list. For each evaluation of the command, the corresponding parameter's values are filled in for parameter keywords (see 5.1.2). This command is the single line equivalent of `$p–template$`. Anything after the specified parameter list is considered part of the line to include. The format is as follows:

```
$p–line$ list_of_parameters code line goes here;
```

There are no format restrictions for the code line, any remaining characters on the command's line are included, the only requirement is that it is separated from list_of_parameters by a single space character.

Chapter 6

Program Structure

The program is written in java and consists of five java classes: Main, TemplateProcessor, CommandResult, Param and Utilities. With the java language and the included maven project setup, the program should be portable and modifiable across different operating systems and IDE's. The maven setup includes a build configuration that produces the executable jar file, which can be run directly from the desktop or command line.

The Main class handles the loading of the settings file and creates the required directories. It matches up the parameters and base-template for each file specified in the settings and begins creating the files by starting the template processor on each base-template.

The TemplateProcessor handles all the instructions in the base template. In order to do so the TemplateProcessor keeps a set of known of parameters and variables for the current template. It walks through the template line by line, filling in the variable values known up till then in the current line and checking if the line contains a command. If a line contains a command, the command is executed and the line is replaced with the result of the command. If there was no command the line (with filled in variables) is simply copied to the output.

A template may specify sub-templates, in this case the processing described above is repeated for the sub-template, but with the known parameters and variables from the current template passed on.

Some commands may instruct the processor to include a section of the template multiple times, once for a specific parameter. In this case the parameter's attributes are loaded as variables, and the section is processed as if it were a sub-template, but with the added parameter variables.

Most commands require one or more lines to be removed beforehand, so a result of a command is represented as a set of new lines and a number of lines to remove with the CommandResult class. Once all the template lines have been processed like this the output is a set of newly generated code, which is sent back to the Main class to be stored in the right output file.

Finally the main class does some basic verification by checking that all braces ({ and }) match up in the generated code. If there is a mismatch the program prints a warning for the user with a line number for the mismatch.

The program uses the Param class as an internal way to represent the parameters specified in the settings file. This class combines all the parameter attributes into an object, which makes it easy to match parameters with certain files or pass them around to various functions. Besides various ways to interpret and output parameters it also includes a simple sorting algorithm for a list of parameters.

The Utility class provides some generic functions that are used by the other classes, such as reading and writing to files in various formats. It also includes functions to find specific line parts such as indentation, as well as a method of counting start / end indicators such as braces to ensure they match up.

Appendix A

Template Examples

A.1 Variables and Sub-Templates

Listing A.1: base template : variables

```
//< This template demonstrates the use of template variables
//< run the program and check the output in examples/variables
//<
//< we define two variables, foo and bar
$var$ \{foo\} 10
$var$ \{bar\} monkey
//<
//< we use the variables here (see the s?)
I saw foo bars

//< variables only take effect after they are defined!
I dont see something
$var$ \{something\} bananas
I do see something!

// now let's see what happens with a subtemplate (see the indentation?)
  $template$ examples/monkeymadness.cgen_template

// and let's check again afterwards in the main template
I still see foo bars
```

Listing A.2: sub-template : monkeymadness

```
// this is the subtemplate

// the variables from the parent are still known!
I still see foo bars

// but we can redefine them locally!
$var$ \{bar\} chimpanzee
But I now see foo bars

// or even use them as part of new variable names!
$var$ \{bar madness\} confusing
Is this chimpanzee madness?
```

Listing A.3: result

```
I saw 10 monkeys

I dont see something
I do see bananas!

// now let's see what happens with a subtemplate (see the indentation?)
  // this is the subtemplate

  // the variables from the parent are still known!
  I still see 10 monkeys

  // but we can redefine them locally!
  But I now see 10 chimpanzees

  // or even use them as part of new variable names!
  Is this confusing?

// and let's check again afterwards in the main template
I still see 10 monkeys
```

Check the console output or the logfile and you'll also notice a warning about the redefinition of the bar value in the sub-template. The program includes many warnings for possibly unintentional scenarios like these.
Assuming the `variables.cgen_template` and `monkeymadness.cgen_template` from this example are in `./templates/examples/` relative to the folder the executable jar file is in, this is how this example would be specified in settings.json. The code generator by default places output files in `./subsystem_name/`, so this would instead create the 'variables' output file in the `./examples/` directory.

Listing A.4: specifying this example in settings.json

```
    {
      "filename" : "../examples/variables",
      "base_template" : "examples/variables.cgen_template"
    },
```

A.2 Parameter Processing Commands

Listing A.5: parameter commands example

```
//< This template demonstrates the use of parameter commands
//< run the program and check the output in examples/paramcommands
#include <stdio.h>

//< First example: initialising all parameters to default value
void initialize() {
  //< p-line : note how often this line is included!
  //< once for each specified parameter!
  $p-line$ [all] p#name = p#defaultValue;
}

//< Second example: reset only the 2 specified parameters
void reset_sensors() {
  $p-line$ [sensor_1|sensor_2] p#name = p#defaultValue;
}

//< Third example: use p-block for multiple lines!
void print_parameter() {
  //< additionally, lets throw in a variable!
  $var$ \{printfunction\} printf
  //< p-block: includes the block once for each parameter!
  $p-block$ [testing_2|testing_4] \{
    // This is easier than typing lots of p-lines!
    printfunction("%u", p#name);
  \}
}

//< final example: use p-template for very large sections
//< or use it to organise smaller sections
void set_parameter() {
  //< we could use the same template for each parameter with :
  //< $p-template$ [all] sometemplate.cgen_template
  //< but we can also use a parameter variables here!
  $p-template$ [all] p#name/set_parameter.cgen_template
  //< note the different template files used for each
  //< and what happens if the file was missing or empty?
  //< (check the logfile / command line output!)
}
```

Listing A.6: parameter commands example

```
// this is the set template for sensor_1
p#name = 500;
```

Listing A.7: parameter commands example

```
// this is the set template for sensor_2
p#name = 1337;
```

The template for testing_2's setter `testing_2/set_parameter.cgen_template` was left empty. The template for testing_4's setter `testing_4/set_parameter.cgen_template` didn't exist.

Listing A.8: result

```
#include <stdio.h>

void initialize() {
  testing_2 = 1000;
  testing_4 = 0xCAFE;
  sensor_1 = 0;
  sensor_2 = 100;
}

void reset_sensors() {
  sensor_1 = 0;
  sensor_2 = 100;
}

void print_parameter() {
    // This is easier than typing lots of p-lines!
    printf("%u", testing_2);
    // This is easier than typing lots of p-lines!
    printf("%u", testing_4);
}

void set_parameter() {
  // Add testing_2 code section here!
  // Add testing_4 code section here!
  // this is the set template for sensor_1
  sensor_1 = 500;
  // this is the set template for sensor_2
  sensor_2 = 1337;
}
```

As can be seen in the example, the p-template command adds a reminder in the output if the template was missing or empty. If the specified template didn't exist, an empty template is automatically created for it as well. This makes it easier to use p-templates to keep large template systems organized, and can be a helpful reminder for future work.