TU DELFT

PQ9 Code Generator Manual

Subsystem software from dynamic templates

Created by:
Erik Mekkes

December 9, 2018

Contents

Chapter 1

Quick Start Guide

1.1 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

1.2 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

1.3 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.

1.4 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 2

Program Settings

- Unfinished -

Chapter 3

Specifying parameters

Chapter 4

Working with templates

4.1 Template Keywords

This chapter lists the pre-defined 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 generation cannot distinguish between intended use, 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.

4.1.1 parameter properties

These keywords are pre-defined to represent parameter properties, they are replaced with the corresponding value's if a specific parameter is passed during the code generation process (see 4.1.2 and 4.1.2). They **can not** be used for any other purpose in a template.

- 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.
- p#dataType : The data type of the parameter.
- p#defaultValue : The default value of the parameter.

4.1.2 commands

These keywords can be used as commands for certain actions within templates. They are identified by the surrounding dollar $ signs.

var

This command can be used within templates to define a variable, the format is as follows:

```
$var$ #variable_name# variable_value
```

The effect is very straightforward, every instance of variable_name 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. The `$var$` command identifier is fixed.

Anything between the first two # characters following the `$var$` command is taken as the variable name, so anything that does not include a # is a valid variable name. Everything between the last $ of the command and the first # of the variable name is ignored. For example `foo_var` is a valid

variable name, but so is `this gets replaced by the value` or `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 4 is an accepted value, but so is `*((uint32_t*)value)` or `I actually want to replace foo_var with this entire sentence`.

Using variable names within a variable value is possible, for example if `foo` and `bar` are defined as variables, `I am foo my value is bar` is accepted as a value for foo. Using the name of a variable in it's value is allowed like above, but it will not be replaced to prevent infinite loops. In the above example, foo would be replaced with `I am foo my value is x` if bar's value was defined as 'x'. Likewise, replacement of variables within variable value is only performed once, corresponding with the order in which the variables are defined within the template.

$template$

This command can be used within templates to include another 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 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.

$p-template$

This command can be used to process a sub-template for a specified set of parameters. The sub-template's lines are included multiple times, once for each specified parameter, replacing this command line. For each evaluation of the sub-template, the respective parameter's values are filled in for parameter keywords (see 4.1.1). The format is as follows:

```
$p–template$ array_of_parameters template_filename
```

The accepted format for the parameter array is:

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

The keyword `all` is an accepted parameter name for this array, which results in processing the template for each of the parameters specified for the parent template.

The parameter names used in parameter_array must be a subset of the available parameters for the output file as specified for the top level base template in settings.json.

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 $p-template$

and $p-line$ commands for parameters while the parent template is being processed for a certain parameter.

$p-line$

This command can be used within templates to process a line for a specified set of parameters. The line is included multiple times, once for each specified parameter. This command is the single line equivalent of `$p-template$`. Anything beyond the specified parameter array is considered part of the line to include. The parameter's values are filled in for any parameter keywords in this line. The format is as follows:

```
$p-line$ array_of_parameters code_line
```

The accepted format for the parameter array is:

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

The keyword `all` is an accepted parameter name for this array, which results in processing the line for each of the parameters specified for the parent template.

The parameter names used in parameter_array must be a subset of the available parameters for the output file as specified for the top level base template in settings.json.

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 array_of_parameters by a single space character. Examples of valid code_line input arguments:

```
p#dataType p#name = p#defaultValue;
```

```
// I want to comment about the keywords p#id, p#name and p#enumValue
```

Chapter 5

Program Structure

Appendix A

Example Templates

A.1 Variables and SubTemplates

A.1.1 Base Template

A.1.2 Sub Template 1

A.1.3 Sub Template 2

A.1.4 Generated Output

A.2 Filling In Parameter Values

A.3 ADB Subsystem : parameters.c

A.3.1 Generated Output