QGen User Guide

Release 25.0w

AdaCore

Dec 13, 2023

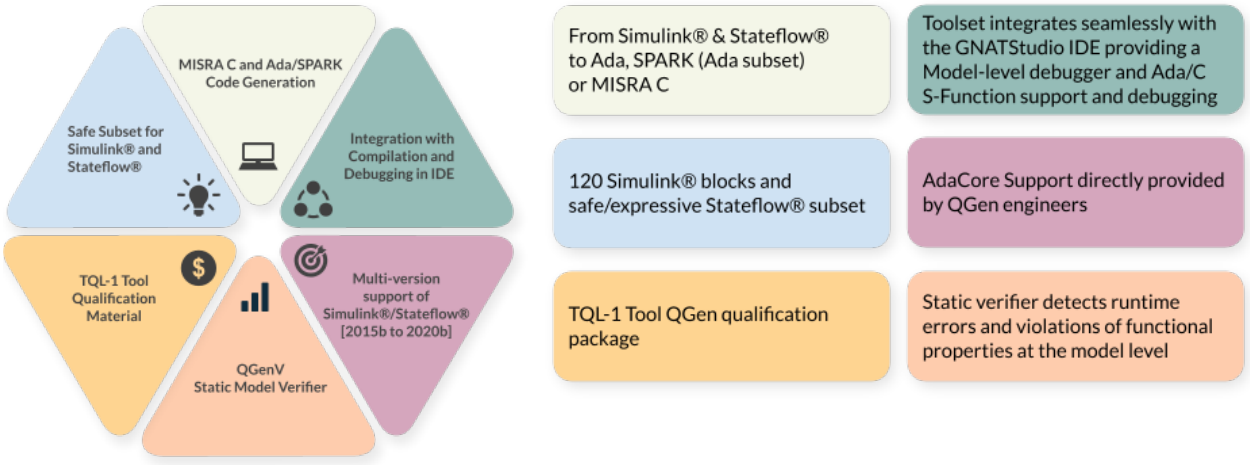
CONTENTS

CHAPTER

ONE

QGEN TOOLSET PRESENTATION

QGen is a Code Generation toolsuite for Simulink® and Stateflow®. Its primary function is code generation from Simulink® and Stateflow® model to C or Ada/SPARK code. The toolset also includes model-level and S-Function debugging with the QGen Debugger add-on.



QGen is officially supported for MATLAB® R2015b up to R2020b. The toolset most likely works on older and more recent versions of MATLAB but there is no guarantee. We continue to support new MATLAB versions as they come out.

The *QGen Code Generator* section presents and details the usage of the QGen Code Generator.

The QGen Code Generator is being qualified at *Tool Qualification Level 1*, which is the highest level of qualification recognized by the FAA and *DO-178C* standard. For more information about the credits that can be claimed by a project using the Qualified tool and pricing information please visit the https://www.adacore.com/qgen/enterprise webpage or contact us through the ticket system within GNATtracker or by sending an email with your account number in the subject #XXXX to qgen@adacore.com.

The supported subset of QGen is made of 120 Simulink blocks with the majority of the settings available for each block, and a set of Stateflow features that are essential for embedded software. A variety of mechanisms for refining the model and tuning code generation is supported such as Simulink.Signal, Parameters, workspace containers (data dictionaries, model workspace), masks, Storage classes … The detail of the QGen supported subset can be found in the *QGen Constraints on Input Models* section.

QGen also comes with full IDE integration, with GNATStudio being used to provide a unique model level debugger, allowing to quickly compile the generated code and analyze its behavior in order to find modeling issues, see the *QGen Model Debugger* and *QGen Model Debugger: Step by Step Tutorial* sections. This also comes with the capability to build and debug S-Functions for C and Ada as well as debugging them while they are simulated in Simulink, see *Calling C or Ada codebases using QGen-SFun block*.

CHAPTER

TWO

INSTALLATION INSTRUCTIONS

2.1 Installing QGen

2.1.1 Windows

To install QGen on Windows simply double-click the setup executable `qgen-<version>-x86[_64]-windows-bin.exe` and follow the instructions to select an installation directory. In the remainder of this guide the chosen installation directory will be refered to as `$QGEN_INSTALL`.

This will enable you to use QGen from the Windows command line, however it is much more convenient to use QGen from the MATLAB environment.

To use QGen from your MATLAB environment, please follow the setup process in *Setting up QGen in MATLAB*.

2.1.2 Linux

To install QGen on Linux extract the archive `qgen-<version>-<platform>-bin.tar.gz` to a temporary location and run the `doinstall` script located in the extracted directory.

The script will guide you through the installation process and will ask you to specify the desired installation directory which will be referred to as `$QGEN_INSTALL` in the remainder of this guide.

To use QGen outside of the MATLAB environment, add the QGen executables location to the `PATH` environment variable by appending the following line to the file `~/.bashrc` or `~/.zshrc` depending on which shell is in use:

```
export PATH=$QGEN_INSTALL/bin:$PATH
```

However it is much more convenient to use QGen from the MATLAB environment. To do so, please follow the setup process in *Setting up QGen in MATLAB*.

2.2 Setting up QGen in MATLAB

QGen provides MATLAB scripts and menus allowing to invoke the toolset directly from the MATLAB/Simulink environment.

To setup QGen for usage within MATLAB, please use the following steps:

1. Start MATLAB.
2. Ensure that a compiler is set for Simulink using the *mex-setup* command. This is needed by QGen because it relies on the *loadlibrary* command that requires a valid compiler to be defined in MATLAB.

3. Run the following command:

```
>> run $QGEN_INSTALL/qgen_install.m
```

OR

navigate to the QGen installation directory `$QGEN_INSTALL` in the MATLAB graphical browser, right-click the script `qgen_install.m` and click **Run**.

After completing the above steps, you will be able to use QGen from your MATLAB environment as explained in the remainder of this guide.

2.3 Uninstalling QGen

To uninstall QGen, first perform the following steps to remove it from the MATLAB environment:

1. Start MATLAB.
2. Run the following command:

```
>> qgen_uninstall
```

OR

navigate to the QGen installation directory `$QGEN_INSTALL` in the MATLAB graphical browser, right-click the script `qgen_uninstall.m` and click **Run**.

Then depending on your operating system, use the following steps to completely remove all QGen files.

2.3.1 Windows

1. Navigate to the QGen installation directory.
2. Execute `uninstall-qgen-<version>.exe` and follow the steps.

2.3.2 Linux

1. Run the following command in your system shell:

```
rm -r $QGEN_INSTALL
```

2. If there are any, remove references to `$QGEN_INSTALL` from your shell startup files `~/.bashrc` or `~/.zshrc`.

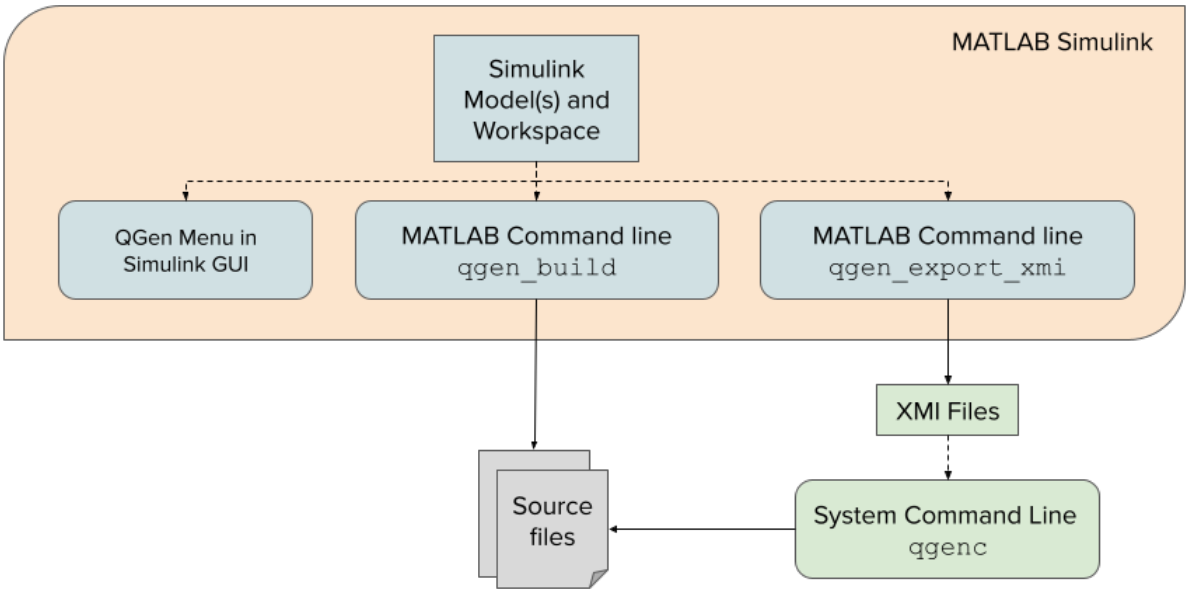
CHAPTER

THREE

QGEN CODE GENERATOR

3.1 Using qgenc

Code generation is performed using the `qgenc` executable which can be invoked in three methods:



1. From the Simulink GUI

   Code generation can be launched directly from within a Simulink model in the MATLAB GUI. This method is the easiest and most straightforward, however it is less flexible than the other methods in terms of the configuration options it provides.

2. From the MATLAB Command Line

   Code generation can be launched by calling the `qgen_build` function in the MATLAB Command Line. This method allows configuring the code generation through arguments passed to `qgen_build`.

3. From the System Command Line

   Code generation can be launched by running the `qgenc` executable from the system command line. This method requires a preliminary execution of the `qgen_export_xmi` MATLAB function to extract relevant information from the Simulink model and serialize it in multiple XMI files, one per model and one for the base workspace, as explained in *Exporting the model information*. Once the files are generated, `qgenc` no longer requires the MATLAB/Simulink environment which makes this method better suited for nightly and regression testing.

3.2 Prerequisites

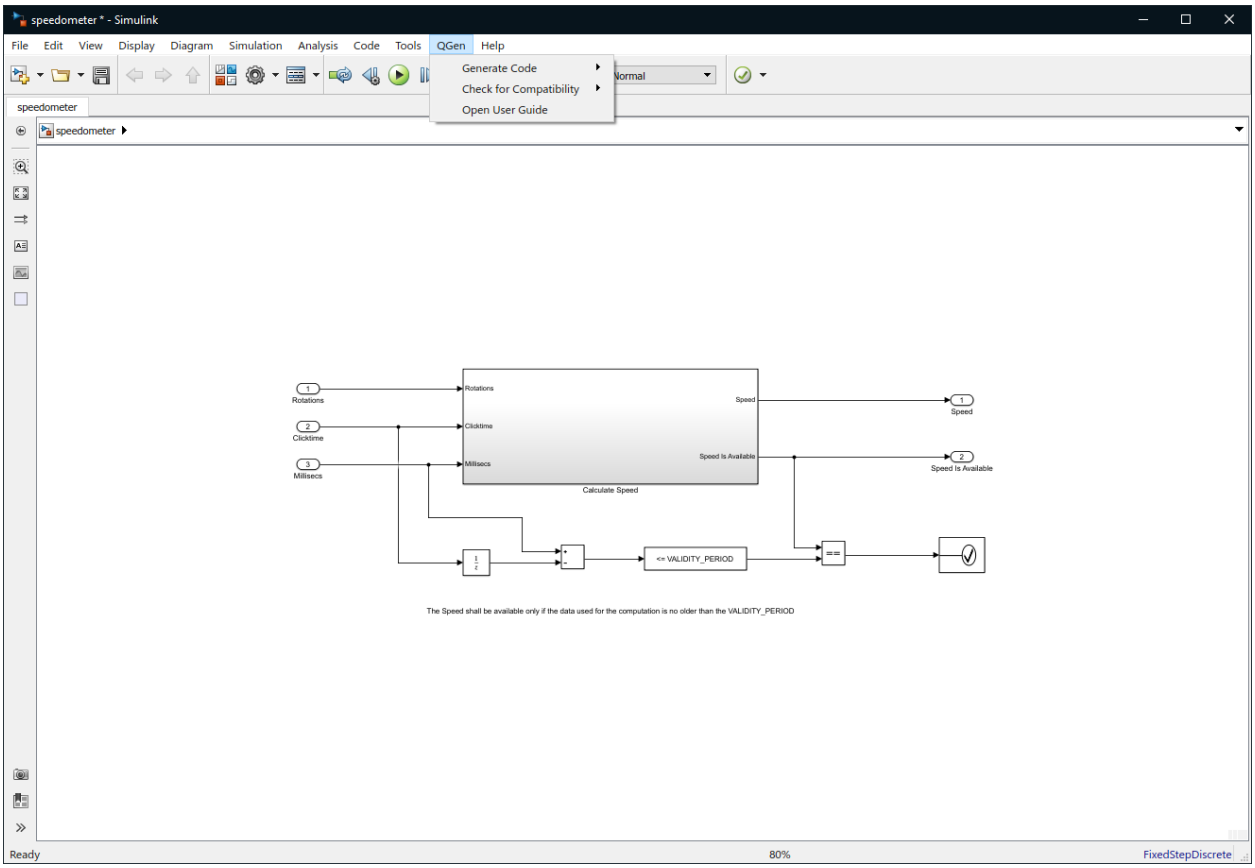
QGen supports models with data values and data types defined in objects that are not directly contained in the models themselves. Such objects can be defined in variables declared either in the MATLAB base workspace, model workspace or data dictionaries, or come from the enumeration classes contained in .m files visible in the MATLAB path.

Before performing code generation with QGen, it is necessary to ensure that the model simulation can be run in Simulink with no errors.

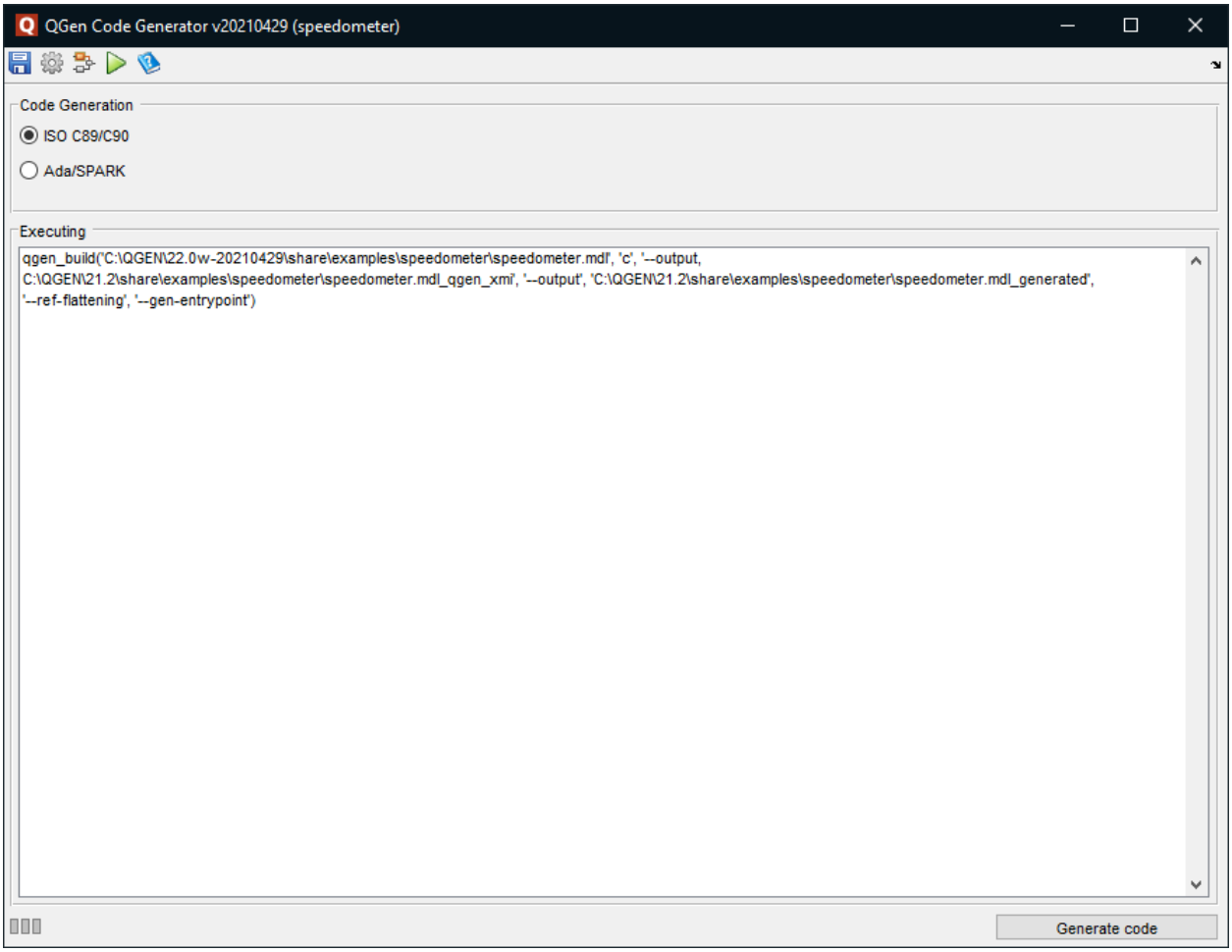
3.3 Generating Code from the Simulink GUI

QGen can be invoked from the Simulink user interface. This requires QGen to be correctly setup in MATLAB following the instructions in *Setting up QGen in MATLAB*.

A QGen menu item is available in the menu bar of any Simulink model.



Clicking on `Generate code` gives an option to generate code for the current model or the active subsystem, and then shows the following menu.



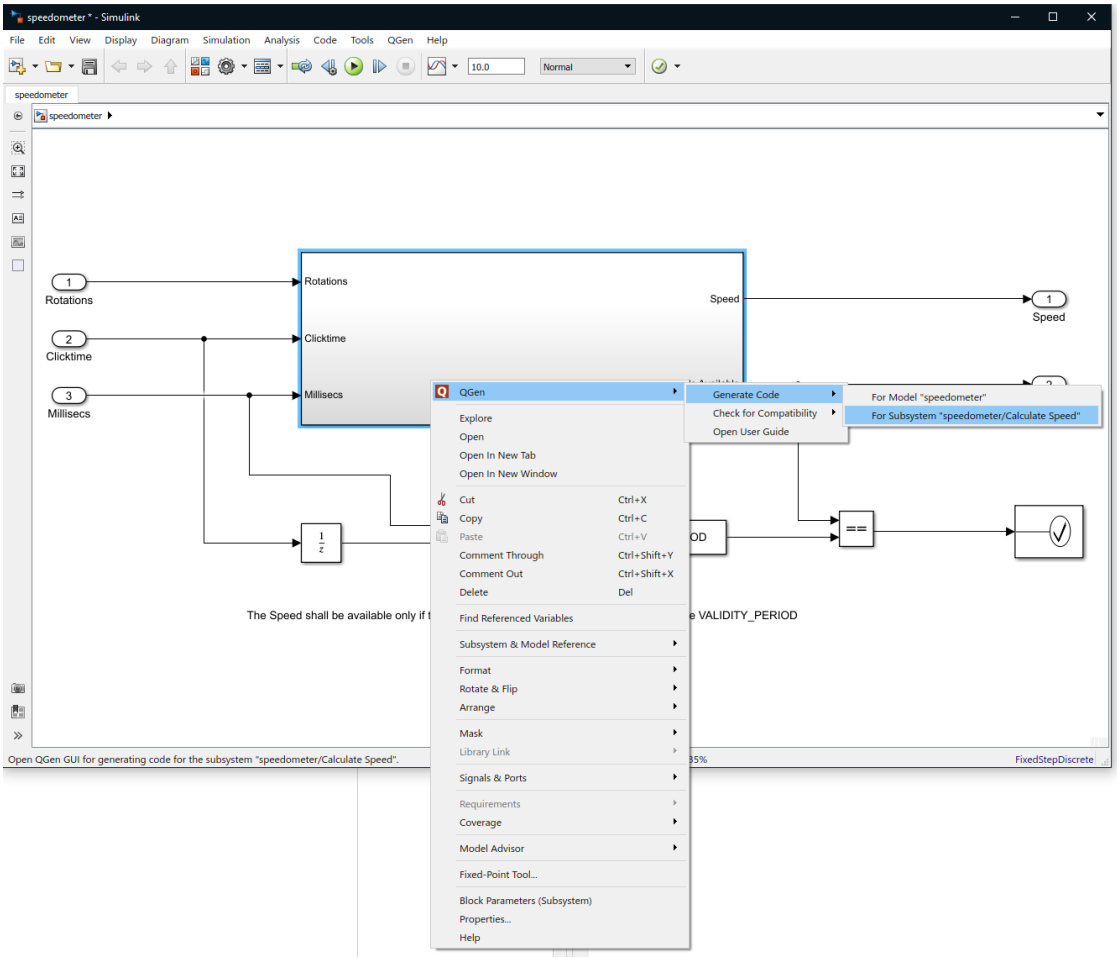
The first row of buttons allows to, in order :

- [Floppy Disk] Save the command, to use it by default for this model
- [Gear Icon] Open the detailed code generation options; see next section.
- [Blocks Icon] Focus or highlight the root block to generate code for
- [Triangle Icon] Run the generation command
- [? Book Icon] Open this User Guide

The Code generation section selects the output language to be C89 (C99 compatible) or Ada (SPARK subset).

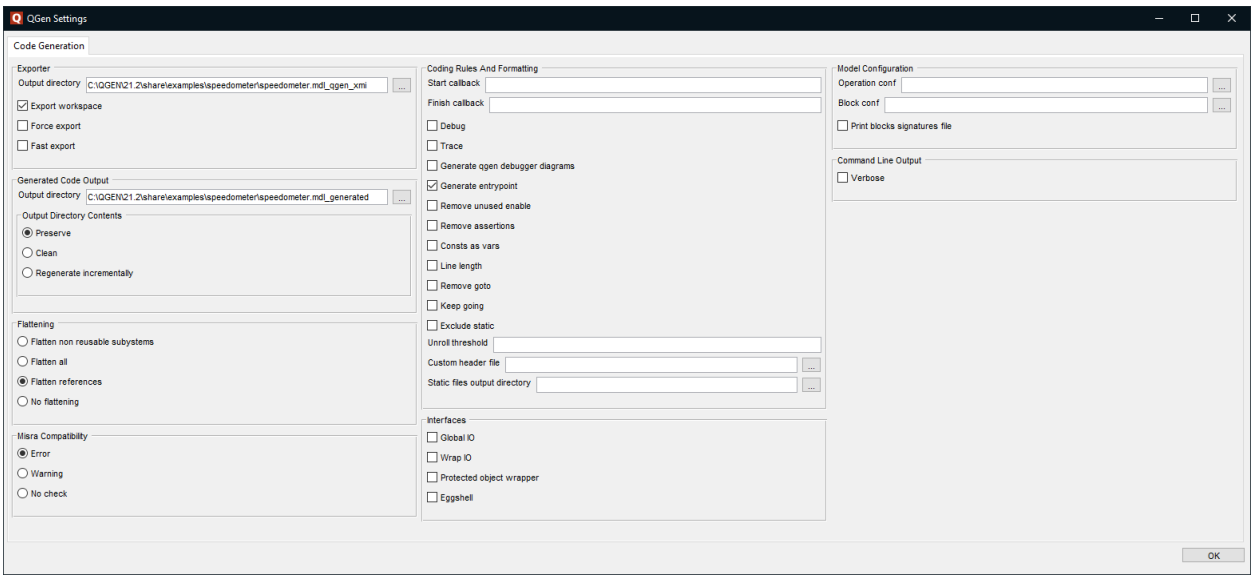
The Executing section shows the *qgen_build* command that is going to run after selecting all the options. It can be edited by hand or through the detailed options panel (see next section). It can be saved in order to be reused in Continuous Integration process or specific scripts.

It is also possible to right-click on a subsystem and execute QGen on a specific subsystem only.



3.3.1 Available options

After clicking on the Gear Icon the following options are available:



Exporter section

- Output Directory: The directory where the files exported from Simulink are stored. If the directory already contains XMI files, the incremental export process will be triggered in order to only re-export models that changed.
- **Export workspace: On by default, causing the workspace data to be exported.** Toggling this off is useful when exporting incrementally and the workspace was previously exported. This can save a lot of time when the workspace data is sizeable and not changing between two subsequent code generation sessions.
- Force export: Off by default. Toggling this on will bypass the incremental export mechanism, in order to fully re-export the entire model. This is useful if changes were not automatically detected, which is rare, but can happen.
- Fast export: Off by default. Toggling this on will cause the exporter to use the Simulink model version parameter to determine whether a model needs to be re-exported, when exporting incrementally. This can increase the export performance but does not provide any guarantee of synchronization between models. Use only if you are sure that none of the model interfaces changed since the last export.

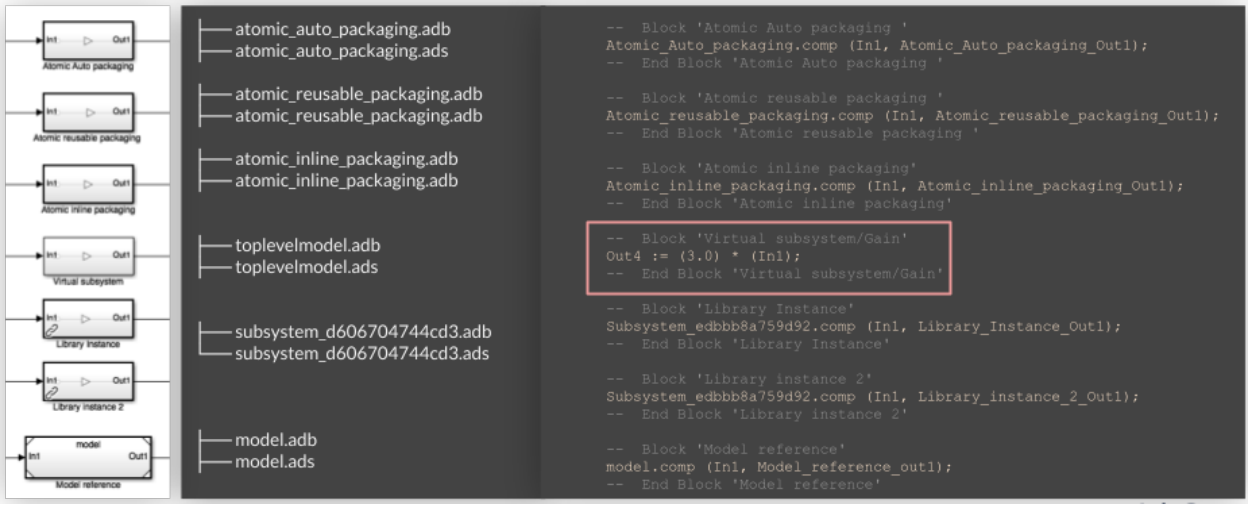
Generated Code Output section

- Output directory: The directory where the generated code is produced.
- Output Directory Contents: *Preserve* (default) will cause the tool to raise an error if the output directory is not empty. This is the default in order to prevent the tool from removing files without a human decision behind it. *Clean* will delete the output directory content before code generation and *Regenerate incrementally* will only regenerate the source code that changed.

Flattening

The following options are available and influence the generated code packaging, in the form of Ada package or C body/header pair.

- Flatten non reusable subsystems: Atomic subsystems do not create their own package if they are not marked as reusable. Use Simulink Function Packaging in the Code Generation options of the Subsystem to specify that property.
- Flatten all: Specifies to flatten all subsystems and models in a single package.
- Flatten references: Default. Each model is flatten in its own package, but library instances are still factorized.
- No flattening: Maximum amount of packages created. Only virtual subsystems do not end up in their own package.



Misra compatibility

Defines the policy applied by the code generator regarding the MISRA Simulink and Stateflow rules enforced, detailed in the *MISRA Simulink Guidelines* section.

- Error: Default behavior, MISRA SLSF violations cause an error.
- Warning: MISRA SLSF violations are specified in a warning.
- No check: MISRA SLSF violations are silently ignored.

Coding Rules and Formatting

- Start callback: Specify MATLAB code that will run before code generation starts.
- Finish callback: Specify MATLAB code that will run after code generation completes.
- Debug: Off by default. When set to on, this flag disables optimizations in the generated code. This impacts the performance of the generated code but provides fine grained traceability between code statements and the model, which can be useful when using the QGen Debugger.
- Trace: Off by default. Generate qdb files in the output directory to allow debugging at the model level using the QGen Debugger and GNAT Studio.
- Generate QGen Debugger diagrams: Off by default. Generate diagram files to allow debugging at the model level using the QGen Debugger and GNATStudio.
- Generate entry point: On by default. Generates an additional package for the top-level model with `init` and `comp` functions, used to call the generated code. This module declares all necessary state variables and provides a simple interface to call from hand-written code. See *Calling the generated code*.
- *Remove unused enable*: Off by default. When set to on, unused code generated for enable blocks will be removed from the generated code. This has the downside of no longer making the code generation modular and disables the capability to generate code incrementally from these sources.
- *Consts as vars*: Off by default. QGen will generate constants as variables allowing the generated code to be tuned at runtime.
- Line length: Forces the generated code to be wrapped to the next line if it exceeds 80 characters.
- *Remove goto*: Off by default. Removes goto statement, only relevant for Stateflow Charts code.

- *Remove assertions*: Remove code for Assertion blocks and their inputs.
- Keep going: Off by default. QGen will not stop on the first error and will uncover as many of them as possible.
- Exclude static: Off by default. QGen will not copy its static files in the output directory. This can be useful if they are provided otherwise in the application integrating the generated code.
- *Unroll threshold*: Modifies the way binary operations on arrays are generated. When an array has less or the same number of elements than the specified threshold, binary operations on the array will be generated as a sequence of elementary binary operations on each element. Otherwise, the binary operations will be expanded to a loop statement iterating over the elements of the array. No value by default, in which case QGen uses 6 as the default value.
- Custom Header File : No value by default. Specify the file containing the comments to be used as header of each generated file.
- Static Files Output Directory: No value by default, the generated code output directory is used in that case. Directory in which the static files provided by QGen are copied

Interfaces

- *Global IO*: Off by default. When set to on, inputs and outputs of the toplevel package functions will be one global variable for each parameter.
- *Wrap IO*: Off by default. When set to on, inputs and outputs of the toplevel package functions will each be provided as a structure.
- *Protected object wrapper*: Off by default. Generate a protected object wrapper (for SPARK/Ada only) to ensure mutual exclusion when changing input signals to execute a computation step.
- Eggshell: Off by default. Consider all root subsystems in "eggshell" models as separate root models when generating code. "Eggshell" models are Simulink models with no interface (Inport or Outport blocks), consisting only of subsystems (with interface or not). Using this pattern is not recommended.

Model Configuration

- Operation conf: Specify a file to use to provide operation replacement information for arithmetic operation. See *Customizing Arithmetic Functions*.
- Block conf: Specify a file to use to provide block replacement information. See *Customizing Block Implementations*.
- Print block signatures file: Off by default. Generate a file named qgen_<root_model>_blocks_signatures.txt to use as a basis for block configuration. See *Customizing Block Implementations*.

Command Line Output

- Verbose: Off by default. When set to on, QGen will output more messages during the code generation process.

3.4 Generating Code from the MATLAB Command Line

QGen can be invoked from the MATLAB command line using the `qgen_build` function. This requires QGen to be correctly setup in MATLAB following the instructions in *Setting up QGen in MATLAB*.

The `qgen_build` function performs the following main steps:

1. Load the given model in Simulink
2. Export the model information using the `qgen_export_xmi` function in a manner similar to *Exporting the model information*.
3. Invoke the `qgenc` executable.

As a result of step 1, Simulink itself is launched. This might take some time as the Simulink model and its referenced models are loaded upon the invocation of `qgen_build`.

3.4.1 Quick Examples

Here are some quick examples on how to invoke the `qgen_build` function.

Given an arbitrary model `mymodel.mdl`, the following invocation generates Ada code in the directory `generated_code/` for the model `mymodel.mdl`. The `--clean` argument instructs QGen to delete all contents of the output directory `generated_code/` before generating code.

```
>> qgen_build ('./mymodel.mdl', 'ada', '-o', './generated_code', '--clean')
```

Let us take the example in the directory `speedometer`, located in the `$QGEN_INSTALL/share/qgen/examples/` directory. This example requires some global variables which are defined in a separate file.

The following exports XMI files into the `qgen_xmi_dir` directory and then generates Ada code in the folder `./generated_code` for `speedometer.mdl` and, by using the `speedometer_def.m` MATLAB file, defines global variables used by the model.

```
>> speedometer_def
>> qgen_build ('./speedometer.mdl', 'ada', '-o, qgen_xmi_dir', '-o', ...
                './generated_code', '--clean')
```

```
>> qgen_build ('./speedometer.mdl', 'ada', '--xmi', '-o', './generated_code', '-m', './speedometer_def.m', '--clean')
[INFO][QGen] 1/2 Processing model "speedometer"
[INFO] Exporting model to xmi speedometer.mdl
[INFO] Exported workspace to QGen_Base_Workspace.xmi
[INFO] Processing model speedometer
[INFO] Exported model to speedometer.xmi
[INFO][QGen] 2/2 Invoking qgenc...
[INFO] /Users/clement/qgen/gms/qgenc/obj/qgenc --from-simulink --language ada -m ./speedometer_def.m --clean "speedometer.mdl_qgen_xmi/speedometer.xmi" --pre-process-xmi --output
"./generated_code"
[WARNING] XMI pre-processing: ignored matlab file ./speedometer_def.m
[INFO] Importing XMI speedometer.xmi
[INFO] Preprocessing model speedometer
[INFO] Finished preprocessing model speedometer
[INFO] Sequencing model speedometer
[INFO] Finished sequencing model speedometer
[INFO] Generating code model speedometer
[INFO] Finished code model generation speedometer
[INFO] Optimising code model speedometer
[INFO] Finished optimising code model speedometer
[INFO] Printing to Ada speedometer
[INFO] Finished Ada printing speedometer

[INFO][QGen] Execution successful
```

In the following invocation the default output directory `speedometer.xmi_generated/` will be used since the `-o` argument was not specified. The `--incremental` argument instructs QGen to keep the current content of the output directory, and to only regenerate for the models that changed this the last generation in that directory.

```
>> qgen_build ('./speedometer.mdl', 'ada', '--incremental')
```

When running from within MATLAB, it is possible to use the `-v` switch in the exporter_options to get the complete output of loading and analyzing the Simulink model. This is particularly useful to detect the use of call backs in models, blocks or port which may disrupt the use of QGen.

If global variables used by the model are defined in several m-files, it is possible to include them all as follows:

```
>> qgen_build ('./my_model.mdl', 'ada', '-v', '--incremental')
```

The command above loads my_model_param.m and my_model_def.m into MATLAB workspace, export the contents of the workspace to file my_model_base_ws.m and executes `qgenc` with command line `qgenc my_model.xmi -l ada --incremental`

3.4.2 `qgen_build` Arguments

The `qgen_build` script has three required inputs and a number of optional parameters:

```
>> qgen_build (<mdl path>, <lang>, <exporter_options> [, <qgen_options>]);
```

- `mdl_path`

  Name of the mdl or slx file optionally with path (absolute or relative), including extension.

- `lang`

  Name of the language. Accepted values are `ada` and `c`.

The exporter options switches should be provided as a single string each value being separated by a comma.

If no exporter_options nor qgen_options are specified exporter_options `-o <modelname>_qgen_xmi` is used by default.

All exporter options are documented in *qgen_export_xmi Arguments*.

Options are all string values. Each option is either a single boolean switch or a switch followed by a value. Options can be passed in an arbitrary order.

Note: Several options correspond to identical or similar options of the `qgenc` executable. As a result, more details on options will be provided in *Generating Code from the System Command Line*.

Note: Options can be passed either as function arguments, or as a cell array of length 1 where the value of the first element is also a cell array containing the list of options.

The allowed *qgen_options* are described below.

`qgen_build` accepts the following boolean switches (no value):

- `--noreuse-flattening`

  Flatten code for subsystems not marked as Reusable function.

- `--ref-flattening, --rf`

  Flatten code for each model.

- `--full-flattening, --ff`

  Flatten code for the entire model.

- `--wrap-io`

  Wrap IO of top-level subsystem to structure.

- `--global-io`

  Generate main subsystem's IO as global variables.

- `--consts-as-vars`

  Always generate system parameters as variables without this switch constants are created.

- `--debug`

  Run in debug mode.

- `--gen-entrypoint`

  Generate an entry module for the top-level model with `init` and `comp` functions. This module declares all necessary states variables and provides a simple interface to call from hand-written code. See *Calling the generated code*.

- `--remove-assertions`

  Remove code for assertion blocks and their inputs.

- `--verbose, -v`

  Print verbose output when exporting the model.

- `--wmisra`

  Treat violations to MISRA Simulink as warnings.

- `--no-misra`

  Accept violations to MISRA Simulink.

- `--eggshell`

  Consider all root subsystems in "eggshell" models as separate root models when generating code. "Eggshell" models are Simulink models with no interface (Inport or Outport blocks), consisting only of subsystems (with interface or not). Using this pattern is not recommended.

- `--po-wrapper`

  Generate a protected object wrapper (for SPARK/Ada only) to ensure mutual exclusion when changing input signals to execute a computation step.

- `--export-ws, -w`

  Export workspace contents to an m-file, ignore MATLAB file passed in the function input.

- `--prettyprint`

  Wrap all lines to 80 characters.

- `--clean, -c`

  Instructs QGen to delete all the content of the output directory before starting the code generation.

- `--incremental, -i`

  Instructs QGen to leave the content of the output directory as is. Generated files will overwrite existing files with the same name, but other existing files will remain untouched. If the input file is an XMI file generated from Simulink, this option performs an incremental code generation that only regenerates the sources that originate from models that changed since the last code generation

Switch-value pairs shall be passed as two arguments. First is the key and the second is value. It is assumed that values never start with `'-'` to distinguish them from other switches

- `--block-conf`

  Specify a file containing the configuration for block libraries. See *Customizing Block Implementations*.

- `--output, -o`

  Specify the code generation target directory. By default the files are placed in `<modelname>_generated`.

- `--export-dir, -e`

  Specify the target directory for the XMI files. By default the files are placed in `<modelname>_qgen_xmi`.

- `--custom-header`

  Specify the file to use as header for all the generated code.

- `--static-output`

  Directory in which the static files provided by QGen are copied.

- `--unroll-threshold N`

  Modifies the way binary operations on arrays are generated. When an array has less or the same number of elements than the specified threshold, binary operations on the array will be generated as a sequence of elementary binary operations on each element. Otherwise, the binary operations will be expanded to a loop statement iterating over the elements of the array. When not specified, QGen uses 6 as the default value.

- `--steps, -s`

  Specify compilation steps (one or more characters from the following list):

  - `p` preprocessor
  - `s` sequencer
  - `g` code model generation
  - `j` remove goto statements (no jump)
  - `o` code model simplification
  - `x` code model expansion
  - `c` code model postprocessing
  - `d` printing to target language
  - `e` export the result of previous step to XMI

  The `--steps` switch determines the transformations that shall be executed by QGenc. The order of switches above corresponds to the order of transformations executed by the tool (except for 'e'). This order is hard-coded and can not be modified by the `--steps` switch.

  Fully optional steps are 'j' and 'o' which can be left out at any time and 'e' which has a specific usage explained below. The other steps can be omitted only in a workflow where either it is desired to stop code generation at certain point or an external tool takes care of omitted intermediate transformations.

  Step 'e' is useful in conjunction with another step only – it dumps the model produced by the step preceding it in the list. For example, to execute the preprocessor and then dump the resulting model to xmi without generating code, use `--steps pe`.

  It is possible to use the 'e' key multiple times, for example to dump to xmi at every step, use: `--steps pesegeoed`.

Executing `qgenc` without the `--steps` switch is equivalent to `--steps psgoxcd` (preprocessor, sequencing, code model generation, code model simplification, code model expansion, code model postprocessing and printing).

It is possible to use both `--debug` and `--steps` switches simultaneously. In that case, `--steps` should be either 'psgxcd' or any combination of those steps with 'e' (e.g. 'psegxcede'). Using the `--debug` switch with the 'o' step is not allowed.

- `--trace`

  Generate qdb files in the output directory to allow debugging at the model level using the QGen Debugger and GNAT Studio.

- `--with-gui`

  Generate diagram files to allow debugging at the model level using the QGen Debugger and GNATStudio.

- `--with-gui-only`

  Generate diagram files for the QGen Debugger without generating code.

- `--subsys`

  Generate code for the subsystem whose fully qualified name is PATH. For example `--subsys MyModel/MySubsystem/AnotherSubsystem`.

Note:

- The selected block must be a Simulink subsystem. The option is not supported for Stateflow charts. However, the selected subsystem can contain Stateflow charts.
- The selected Simulink subsystem can have a dedicated trigger or enable port (i.e. contain a Trigger or Enable block at the root level).
- The selected Simulink subsystem must not have function call signals entering or leaving the subsystem's boundary through other ports (i.e. input or output ports).

- `--arith-conf`

  Specify a file containing the configuration for arithmetic operations. See *Customizing Arithmetic Functions*.

- `--blocks-signatures`

  Generate a file named qgen_<root_model>_blocks_signatures.txt to use as a basis for block configuration. *Customizing Block Implementations*.

3.5 Generating Code from the System Command Line

`qgenc` is the core executable that performs code generation. It runs completely outside the MATLAB/Simulink environment but requires representation of the model in a set of XMI files generated from the model within MATLAB/Simulink.

3.5.1 Exporting the model information

A set of XMI files is produced from within Simulink by executing the `qgen_export_xmi` function which is available in MATLAB after setting up QGen in MATLAB as per the instructions in *Setting up QGen in MATLAB*. One file is generated for the base workspace, called QGen_Base_Workspace.xmi and one file will be generated per model referenced (including the root).

During generation of the files warning and errors will be displayed, they are not necessarily blocking QGen from executing properly.

Note: The export phase will by default only re-export models that changed since the last export in the given output directory. This means that the first export can be long but if the model is architectured with multiple model references and only some are modified then subsequent incremental exports will be faster.

3.5.2 `qgen_export_xmi` Arguments

The function has one mandatory argument, the name of the Simulink model, and several optional arguments to tune its behaviour:

```
qgen_export_xmi (<mdl_path> [, <options>])
```

- `mdl_path`

  Name of the mdl or slx file optionally with an absolute or relative path, including extension (`.mdl`, or `.slx`).

Options are all string values. Each option is either a single boolean switch or a switch followed by a value. Options can be passed in an arbitrary order.

Note: Options can be passed either as function arguments, or as a cell array of length 1 where the value of the first element is also a cell array containing the list of options.

The accepted boolean switches are:

- `--verbose, -v`

  Print verbose output when exporting the model.

- `--force, -f`

  Re-export the entire model discarding previously generated XMI files in the output directory.

- `--fast-export, -fe`

  Only look at the model version to determine whether a model has to be re-exported or not. Export will be faster but this can cause faulty code generation when non-modified model have datatypes that depend on other modified models.

- `--no-simulink-sequencing, -ns`

  Disable the use of Simulink Debugger to retrieve the block sequencing, see more detail on the next option below. This option will provide a faster export process at the risk of a deviation between the generated code and the Simulink sequencing.

- `--force-simulink-sequencing, -fs`

  Force the use of Simulink Debugger to retrieve block sequencing, this will open each referenced model in Simulink and actively focus it. Without those options QGen uses the Debugger by default on subsystems that contain DataStores or Function-Call Susbsystem that can cause complex sequencing patterns. The default option is the best compromise between a fast export and accurate sequencing data.

- `--no-workspace-export, -nw`

  Do not export the workspace.

- `--export-unsupported, -eu`

  Force the export of blocks not supported by QGen. This option should not be used when code generation is intended. It is provided for translation of XMI to other languages and for advanced compatibility checking.

Switch-value pairs shall be passed as two arguments. First is the key and the second is value. It is assumed that the value does not start with `'-'` to distinguish it from other switches.

- `-o, --output`

  Specify the output directory where generated XMI files are stored. By default files are place in <model-name>_qgen_xmi.

3.5.3 `qgen_export_xmi` Examples

Example 1: Exporting the XMI files of `test.slx` with default parameters

```
qgen_export_xmi ('test.slx')
```

Example 2: Exporting the XMI files of `test2.mdl` in the `XMI_test2` folder without exporting the workspace.

```
qgen_export_xmi ('./test2.mdl', '-o', 'XMI_test2', '-nw')
```

3.5.4 `qgenc` Arguments

`qgenc` is invoked on the system command line as follows:

```
qgenc FILE [switches]
```

`FILE` is an .xmi file exported within MATLAB compliant with the *Geneauto* metamodel as available in `share/qgen/plugins/geneauto/geneauto.ecore`.

The following command lists all command line arguments supported by `qgenc`:

```
qgenc --help
```

Mandatory Arguments

For code generation, there are three mandatory arguments:

```
qgenc FILE [switches] --language ada|c
```

- `FILE` is the root model file exported in xmi format.
- `--language, -l VALUE`

  `VALUE` can be either `ada` or `c`. If the `--language` switch is not provided, the tool will not produce any code.

Input and Output Data

- `--output, -o DIR`

  Specifies an output directory for the generated code. If this argument is not specified, code is generated in `<FILE>_generated` by default. If the output directory is not empty, then one of the switches `--clean` or `--incremental` must be provided.

- `--clean, -c`

  Instructs QGen to delete all the content of the output directory before starting the code generation.

- `--incremental, -i`

  Instructs QGen to leave the content of the output directory as is. Generated files will overwrite existing files with the same name, but other existing files will remain untouched.

When passing an .mdl or a .slx file, the following additional switches are available:

- `--trace`

  Generate qdb files in the output directory to allow debugging at the model level using GNAT Studio.

MISRA Constraints Options

QGen automatically checks that the input model conforms to selected MISRA Simulink constraints. QGenc should not be used as a MISRA checker tool. The checks are limited to constructs which directly influence the code generation policy. For full check one should use a dedicated MISRA checker.

The following flags control the implemented MISRA checks:

- `--no-misra`

  Disable checks for MISRA Simulink constraints.

- `--wmisra`

  Treat violations of MISRA Simulink as warnings.

Code Generation Options

- `--block-conf`

  Specify a file containing the configuration for block libraries. See *Customizing Block Implementations*.

- `--debug`

  Generate non-optimized code

- `--gen-entrypoint`

  Generate an entry module for the top-level model with `init` and `comp` functions. This module declares all necessary states variables and provides a simple interface to call from hand-written code. See *Calling the generated code*.

- `--from-simulink`

  Indicates that `qgenc` was executed from the MATLAB command line. This instructs QGen to produce block references in error messages as Simulink hyperlinks.

- `--noreuse-flattening`

  Use Simulink Function Packaging for subsystem code generation.

- `--ref-flattening, --rf`

  Flatten code for each model.

- `--full-flattening, --ff`

  Specifies to flatten all non-virtual subsystems.

- `--global-io`

  Generates the function parameters of the main module as global variables in a separate module.

- `--wrap-io`

  Encapsulate formal parameters in Ada record types or C structs.

- `--consts-as-vars`

  Generates all constants defined in .m files as variables.

- `--remove-assertions`

  Remove code for assertion blocks and their inputs.

- `--steps, -s VALUES`

  Specifies the code generation steps, which are:

  - `p` preprocessor
  - `s` sequencing
  - `g` code model generation
  - `o` code model optimization
  - `x` code model expansion
  - `j` removal of goto statements
  - `c` code model postprocessing
  - `d` printing
  - `e` export to xmi

For example, to execute the preprocessor and then dump the resulting model to xmi without generating code, use `--steps pe`.

It is possible to use the same code generation step multiple times, for example to dump to xmi at every step, use: `--steps pesegeoed`.

Executing `qgenc` without this switch is equivalent to `--steps psgxcd` (preprocessor, sequencing, code model generation, code model expansion, code model postprocessing and printing).

- `--unroll-threshold N`

  Modifies the way binary operations on arrays are generated. When an array has less or the same number of elements than the specified threshold, binary operations on the array will be generated as a sequence of elementary binary operations on each element. Otherwise, the binary operations will be expanded to a loop statement iterating over the elements of the array. When not specified, QGen uses 6 as the default value.

- `--verbose, -v`

  Display debug level messages

- `--static-output`

  Directory in which the static files provided by QGen are copied.

- `--custom-header`

  Specify the file to use as header for all the generated code.

- `--all-variants`

  Generate code for all Model Reference Variants and an If statement choosing which Variant to execute using the Variant Condition from Simulink.

- `--arith-conf`

  Specifies a file with transformations for arithmetic functions. See *Customizing Arithmetic Functions*.

- `--no-jump`

  Removes goto statements.

- `--prettyprint NUM, -pp NUM`

  Format lines to be at most NUM characters long.

- `--remove-unused-enable`

  Unused code generated for enable blocks will be removed from the generated code. This has the downside of no longer making the code generation modular and disables the capability to generate code incrementally from these sources.

- `--subsys PATH`

  Generate code for the subsystem whose fully qualified name is `PATH`. For example `--subsys MyModel/MySubsystem/AnotherSubsystem`.

Note:

- The selected block must be a Simulink subsystem. The option is not supported for Stateflow charts. However, the selected subsystem can contain Stateflow charts.
- The selected Simulink subsystem can have a dedicated trigger or enable port (i.e. contain a Trigger or Enable block at the root level).
- The selected Simulink subsystem must not have function call signals entering or leaving the subsystem's boundary through other ports (i.e. input or output ports).

- `--eggshell`

  Consider all root subsystems in "eggshell" models as separate root models when generating code. "Eggshell" models are Simulink models with no interface (Inport or Outport blocks), consisting only of subsystems (with interface or not). Using this pattern is not recommended.

3.5.5 qgenc Examples

To incrementally generate C code for `myModel.mdl` exported into `myModel_qgen_xmi` folder inside `myFolder` and using full flattening:

```
qgenc myModel/qgen_xmi/myModel.xmi --incremental --language c --output myFolder --full-
↪flattening
```

Note: The XMI file `myModel.xmi` is generated by running the following in MATLAB prior to invoking `qgenc`:

```
qgen_export_xmi('./myModel.mdl')
```

For more information on the generation of XMI files see *Exporting the model information*.

To do the same, but dump the xmi after the import:

```
qgenc myModel.xmi --incremental --language c --output myFolder --steps epsgd --full-
↪flattening
```

To just run the importer and dump the xmi

```
qgenc myModel.xmi --output myFolder --steps e
```

3.6 Errors and Warnings

When investigating errors and warnings issued by the QGen toolset, first make sure that the model is consistent and that simulation can be run in Simulink. Often errors issued by QGen are due to inconsistencies in the model that should be fixed within Simulink before invoking QGen tools.

The detailed list of errors and warnings issued by the QGen tools is given in *Appendix A: Errors and warnings*.

3.7 Calling the generated code

To call the code generated from hand-written code you should use the `--gen-entrypoint` switch to generate a module called `qgen_entry_[modelname]`.

This module contains two exported functions `init` and `comp` (prefixed by the model name in C) that you should call in that order.

The `init` function corresponds to the Simulink initialization phase and initializes the values for static variables across all the code, including the code coming from all referenced models.

The `comp` function corresponds to the execution of one Simulink iteration step and is typically called inside a loop.

There are several ways how data can be passed to and from the comp when executing it.

3.7.1 Default interface

By default an input argument is generated for each input port and output argument for each output port in comp function.

```
extern void qgen_entry_CodeOrder_init (void);
extern void qgen_entry_CodeOrder_comp
  (GAREAL const In1,
   GAREAL const In2,
   GAREAL* const Out1,
   GAREAL* const Out2);
```

```
package qgen_entry_CodeOrder is

   procedure init;

   procedure comp
      (In1 : Long_Float;
       In2 : Long_Float;
       Out1 : out Long_Float;
       Out2 : out Long_Float);

end qgen_entry_CodeOrder;
```

3.7.2 Wrap IO

The --wrap-io switch tells qgen to encapsulate parameters corresponding to input and output ports into a datastructure. The list of IO parameters of the comp function is replaced with a single input-output argument.

```
typedef struct {
   GAREAL In1;
   GAREAL In2;
} qgen_entry_CodeOrder_comp_Input;
typedef struct {
   GAREAL Out1;
   GAREAL Out2;
} qgen_entry_CodeOrder_comp_Output;

extern void qgen_entry_CodeOrder_init (void);
extern void qgen_entry_CodeOrder_comp
  (qgen_entry_CodeOrder_comp_Input const* const Input,
   qgen_entry_CodeOrder_comp_Output* const Output);
```

```
package qgen_type_wrap_io_qgen_entry_CodeOrder is

   type qgen_entry_CodeOrder_comp_Input is record
      In1 : Long_Float;
      In2 : Long_Float;
   end record;
```

(continues on next page)

(continued from previous page)

```
   type qgen_entry_CodeOrder_comp_Output is record
      Out1 : Long_Float;
      Out2 : Long_Float;
   end record;

end qgen_type_wrap_io_qgen_entry_CodeOrder;

package qgen_entry_CodeOrder is

   procedure init;

   procedure comp
      (Input : qgen_entry_CodeOrder_comp_Input;
       Output : out qgen_entry_CodeOrder_comp_Output);

end qgen_entry_CodeOrder;
```

3.7.3 Global IO

When calling qgen with `--global-io` switch the parameters are replaced with global variables. Input values are expected to be written to these variables before calling the `comp` function and the output values are written to global variables corresponding to output ports. Each of the generated IO variables can be replaced with a variable imported from an external module. See *Importing variables and constants defined in external modules* for a detailed description of this scenario.

```
CodeOrder_State CodeOrder_memory;
GAREAL qgen_entry_CodeOrder_comp_In1;
GAREAL qgen_entry_CodeOrder_comp_In2;
GAREAL qgen_entry_CodeOrder_comp_Out1;
GAREAL qgen_entry_CodeOrder_comp_Out2;

extern void qgen_entry_CodeOrder_init (void);
extern void qgen_entry_CodeOrder_comp (void);
```

```
package qgen_entry_CodeOrder_io is
   CodeOrder_memory : CodeOrder_State;
   qgen_entry_CodeOrder_comp_In1 : Long_Float;
   qgen_entry_CodeOrder_comp_In2 : Long_Float;
   qgen_entry_CodeOrder_comp_Out1 : Long_Float;
   qgen_entry_CodeOrder_comp_Out2 : Long_Float;

end qgen_entry_CodeOrder_io;

package qgen_entry_CodeOrder is

   procedure init;

   procedure comp;

end qgen_entry_CodeOrder;
```

3.7.4 Protected object wrapper

The `--po-wrapper` switch works in conjunction with the `--global-io` and instructs QGen to put the global IO variables within the private part of an Ada protected object. This switch has no effect when generating code for C.

Access to the input/output values is granted through protected procedures as setters for the different input signals, and protected functions as getters for the output signals.

The input signals are set asynchronously by the setter procedures, and then the `comp` procedure uses the latest values for the input signals to compute the new output signals, that can be asynchronously read by the getter functions.

```
package qgen_entry_CodeOrder is

  procedure init;

  procedure comp
      (Out1 : out Long_Float;
      Out2 : out Long_Float);

  protected po is

      procedure comp
        (Out1 : out Long_Float;
          Out2 : out Long_Float);

      procedure set_In1 (In1 : Long_Float);

      procedure set_In2 (In2 : Long_Float);

      function get_Out1 return Long_Float;

      function get_Out2 return Long_Float;

  private
      qgen_entry_CodeOrder_comp_In1 : Long_Float;
      qgen_entry_CodeOrder_comp_In2 : Long_Float;
      qgen_entry_CodeOrder_comp_Out1 : Long_Float;
      qgen_entry_CodeOrder_comp_Out2 : Long_Float;
  end po;

end qgen_entry_CodeOrder;
```

3.7.5 Passing Model Arguments

In Simulink, users may parameterize block parameter values in different instances of a referenced model by defining model arguments in that model's model workspace. Such arguments are typically defined as MATLAB variables and used in block parameters of blocks contained in the model. Users may then customize the model arguments values when referencing the model through a ModelReference block.

There are two cases to consider for the code generated by QGen:

1. Model arguments defined in referenced models

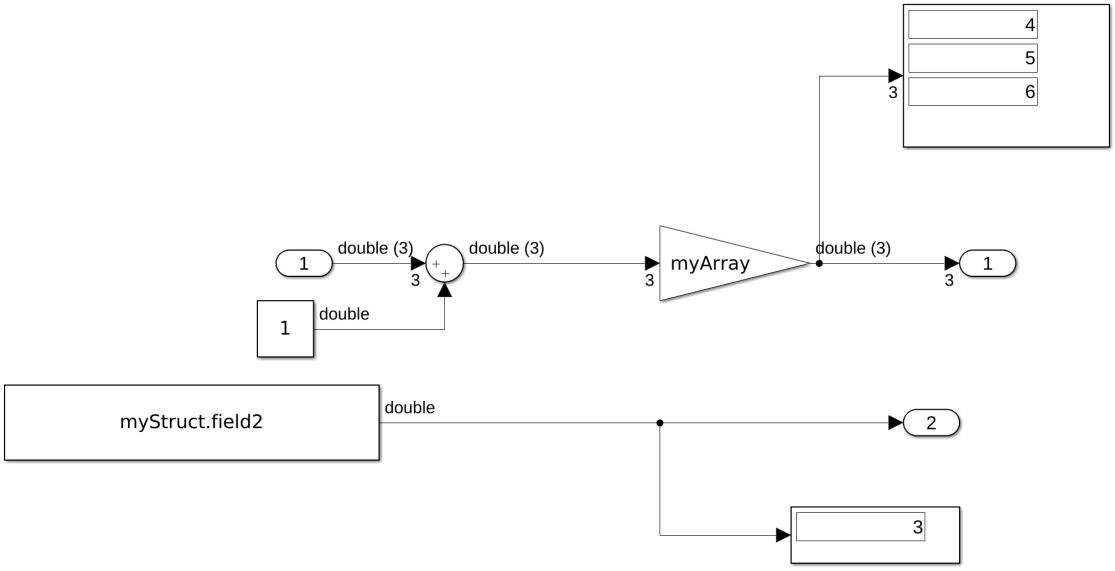
   When model arguments are defined in models referenced at any level, but different from the top-level input model, QGen is able to generate code that will initialize the model argument variables to the specified

values for each instance and reference them in the block parameters where they are used in the referenced model. There is no action required from the user to integrate the generated code.

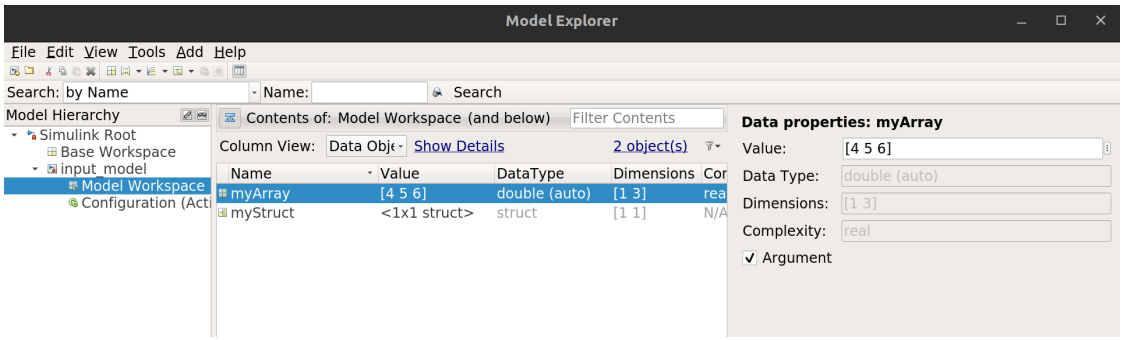
2. Model arguments defined for the top-level input model

   When model arguments are defined in the top-level input model, QGen generates code that references the model argument variables. However, they are not initialized, since it is not possible to know their values in the context where the code is being integrated. The user must initialize all its model argument variables before calling the `init` function, or preferably, the `qgen_entry_[modelname]_init` function.

   For example, consider the following input model defining `myArray` and `myStruct` variables as model arguments:



where the model arguments are defined in the model workspace as:



The generated `init` function without the `--gen-entrypoint` option is shown below.

```
void input_model_initStates
  (GAREAL const myArray[3],
  myStruct_struct const* const myStruct,
  input_model_State* const State)
{
  GAUINT8 i;
```

(continues on next page)

(continued from previous page)

```
  for (i = 0; i <= 2; i++) {
      State->myArray[i] = myArray[i];
  }
  State->myStruct = *myStruct;
}
```

The model arguments are generated as formal parameters of the `init` function and then assigned to the state structure containing memories for the model.

When using the `--gen-entrypoint` option, QGen also generates formal parameters for the model arguments of the `qgen_entry_input_model_init`, which passes them to the `init` function.

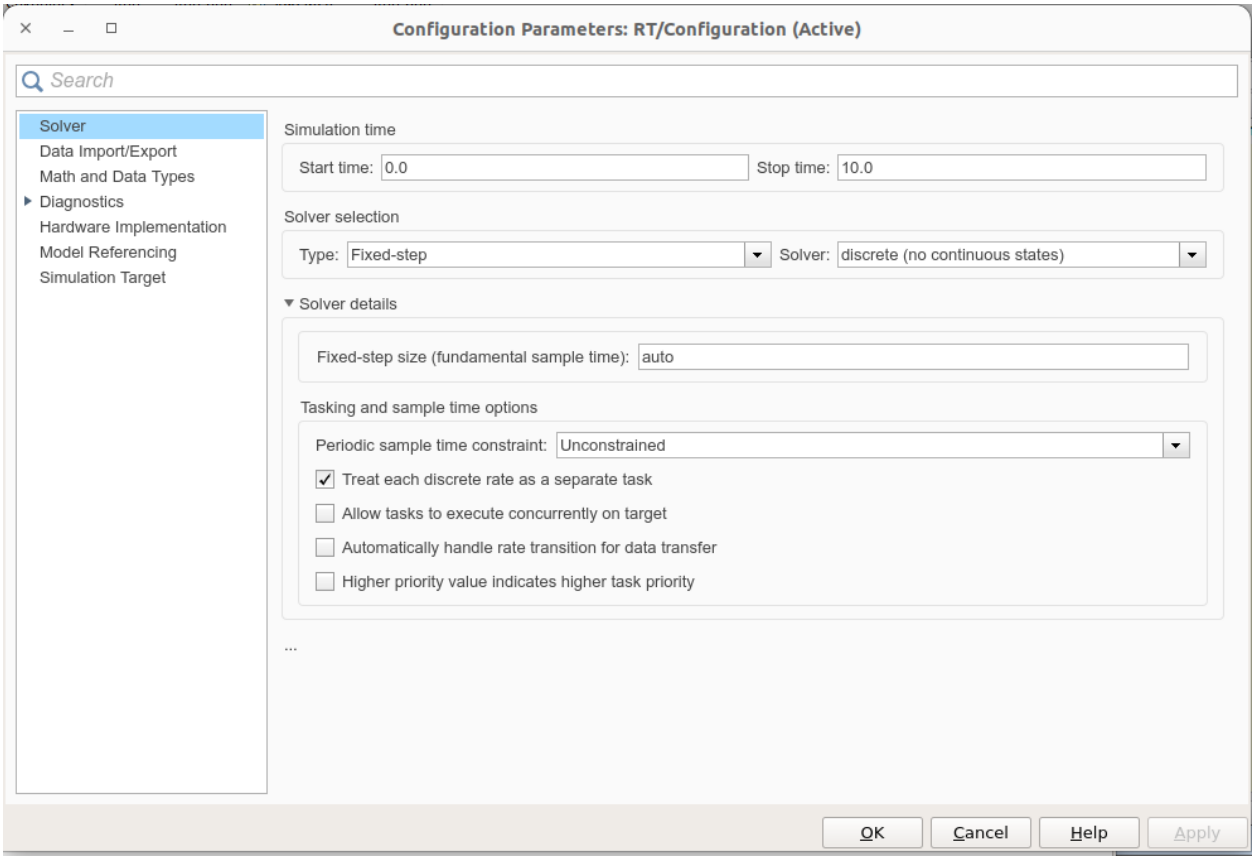
```
void qgen_entry_input_model_init
  (GAREAL const myArray[3],
  myStruct_struct const* const myStruct)
{
  input_model_initStates (
      myArray,
      myStruct,
      &input_model_memory);
}
```

When integrating the generated code, **the user must then ensure that model arguments are completely and correctly initialized before calling the above function**:

```
#include "qgen_entry_input_model.h"

  typedef struct {
    GAREAL field1;
    GAREAL field2;
  } myStruct_struct;

  myStruct_struct myStruct_val;
  const GAREAL myArray_val[3] = {1.01E+02, 2.01E+02, 3.01E+02};

  myStruct_val.field1 = 1.1;
  myStruct_val.field2 = 2.1;

  qgen_entry_input_model_init (
      myArray_val,
      &myStruct_val);
```

3.8 Multirate models

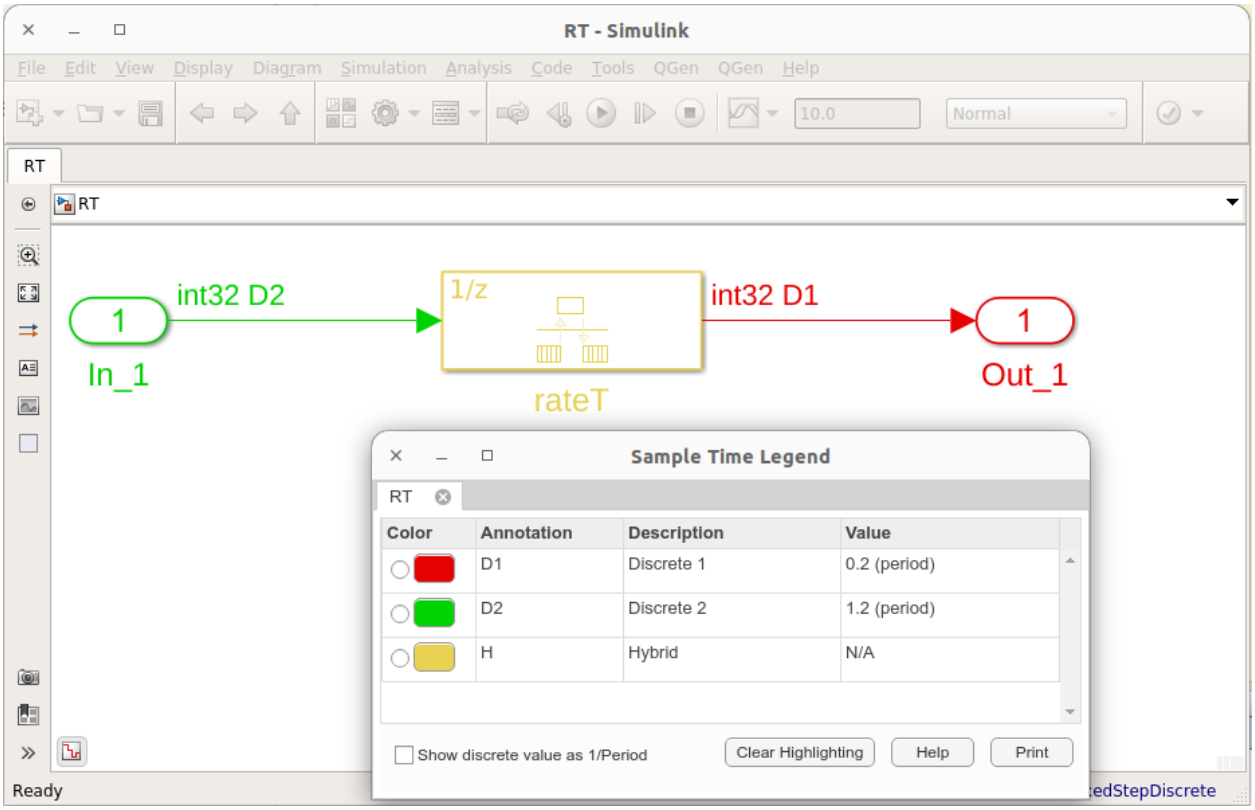
QGen supports code generation for multirate models in multitasking mode i.e. the models where the flag “Treat each discrete rate as a separate task” is set.



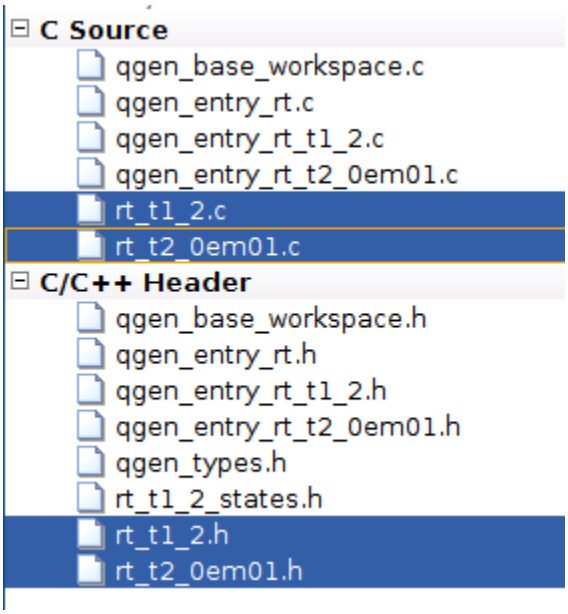
For such a model, QGen groups blocks with the same period and offset and teats each of these groups as a root-level subsystem (also referred to as “task subsystem” in this document) with name:

<model name>_T<period>[_O<offset>]

Each of these new task subsystems will be converted to a target language module (a task module). For instance the model below with two unique rates



will be converted to two modules as follows:



Note, that the period and offset are encoded in scientific notation where commas are replaced with underscores, "-" with "m" and exponent "0" is dropped. In this example

- "T2_0em01" corresponds to "T=2.0e-01"
- "T1_2" corresponds to "T=1.2e00"

If the exponent was positive e.g for period T=20 the corresponding notation in module name would be "T2_0e01".

The patterns for generating module functions and entrypoint modules are the same as with the single root subsystem of a single-rate model:

```
#ifndef RT_T1_2_H
#define RT_T1_2_H
#include "rt_t1_2_states.h"
#include "qgen_types.h"

extern void RT_T1_2_initStates
  (RT_T1_2_State* const State);
extern void RT_T1_2_comp
  (GAINT32 const In_1,
   RT_T1_2_State* const State);
extern void RT_T1_2_up
  (RT_T1_2_State* const State);

#endif
/*  @EOF  */
```

Also, an entrypoint module is generated for each of the task modules in case the *–gen-entrypoint* switch is used.

```
#include "qgen_entry_rt_t1_2.h"

RT_T1_2_State RT_T1_2_memory;
void qgen_entry_RT_T1_2_init (void)
{
   qgen_base_workspace_init ();
   RT_T1_2_initStates (&RT_T1_2_memory);
}
void qgen_entry_RT_T1_2_comp
  (GAINT32 const In_1)
{
   RT_T1_2_comp (
      In_1,
      &RT_T1_2_memory);
   RT_T1_2_up (&RT_T1_2_memory);
}
```

Composing a scheduler that executes the generated tasks with correct rate is the responsibility of the user. The scheduler shall:

- Execute once the **_init* functions of all generated modules (here and in the following bullet points, "all generated modules" refer to entrypoint modules when *–gen-entrypoint* is used, modules generated from individual subsystems otherwise)
- Execute each of the task module **_comp* (and **_up* if *–gen-entrypoint* was not used) at the frequency noted in the name of the module.

An example of a simple interleaving scheduler for the given example would be:

```
20  static GAUINT64 tick;
21
22  /* RUN THIS ONCE BEFORE EXECUTING THE COMPUTE STEPS */
23  void Run_Init_Step (void)
24  {
25     qgen_entry_RT_init ();
26     qgen_entry_RT_T1_2_init ();
27     qgen_entry_RT_T2_0Em01_init ();
28     tick = 0;
29  }
30
31  /* RUN THIS AT THE BASE RATE OF THE MODEL */
32  void Simulation_Run_Comp_Step (void)
33  {
34     static GAINT32 In_1;
35     static GAINT32 Out_1;
36
37     /* ASSIGN INPUT VALUES HERE */
38
39     qgen_entry_RT_comp ();
40     if (qgen_rem_gauint64 (tick, 6) == 0) {
41        qgen_entry_RT_T1_2_comp (In_1);
42     }
43     qgen_entry_RT_T2_0Em01_comp (&Out_1);
44     (tick)++;
45
46     /* READ OUTPUT VALUES HERE */
47  }
```

The function Run_Init_Step calls all the init functions from the entrypoint modules:

- qgen_entry_RT_init – initializes the workspace variables
- qgen_entry_RT_T1_2_init – state variables for the blocks running at rate 1.2
- qgen_entry_RT_T2_0Em01_init – state variables for the blocks running at rate 0.2

The function Simulation_Run_Comp_Step shall be executed at the base rate of the model (0.2 in this case). The global variable *tick* controls the execution of each individual task:

- qgen_entry_RT_T1_2_comp – shall be executed at rate 1.2 i.e on every 6th step.
- qgen_entry_RT_T2_0Em01_comp – shall be executed at each step.

Warning: Importantly, to achieve the simulation results that match with Simulink simulation at each step of execution, the tasks processing inputs should be executed before those processing outputs.

3.9 User-specific Headers

It is possible to add user-specific headers by modifying the files `header.ada_header` and `header.c_header` in the directory `share/qgen/config` of your QGen installation.

It is also possible to specify a file to be used as header using the *–custom-header* command line option or *Custom Header File* field.

3.10 Bus Signals

Bus is a concept for composing composite signals in Simulink. Each signal in a bus is referred as **bus element**. There are two types of buses – **virtual** and **non-virtual**.

Virtual buses refer to a collection of several grouped signals. Grouping does not imply an underlying composite datastructure. However, depending on bus contents a datastructure may be defined implicitly. Also the user may choose to define an explicit datastructure.

Homogeneous virtual buses are buses where all the leaf elements have the same base type. Typically Simulink treats this signal as an array of the common base type.

Non-homogeneous virtual buses are buses with mixed leaf element data types, or homogeneous base types but with some leaf elements that are non-virtual buses. Elements of these buses can be accessed inside of bus-capable blocks (see below), but one should not assume a specific datastructure.

Non-virtual buses must correspond to a pre-defined structure type. These structure type definitions are stored as Bus Objects in Simulink and every non-virtual bus must reference a bus object.

Bus elements may salars or arrays of native Simulink data types or other buses. We refer to bus signals containing one or more elements with bus data type as **nested buses**.

3.10.1 Bus capable blocks

Simulink supports bus signals for the following blocks:

Block	Input	Output
Assignment(nonvirtual buses)	OK	OK
Bus Assignment	OK	OK
Bus Creator	OK	OK
Bus Selector	OK	OK
Constant (nonvirtual buses)		OK
Data Store Memory (nonvirtual buses)	Has no ports, can store bus signals	
Data Store Read (nonvirtual buses)		OK
Data Store Write (nonvirtual buses)	OK	
Delay	OK	OK
Interpolation Using Prelookup	OK	
Memory	OK	OK
Merge	OK	
Multiport Switch	OK	
Prelookup	OK	
Rate Transition		OK
Signal Conversion	OK	OK
Signal Specification	OK	OK
Switch	OK	OK
Unit Delay	OK	OK
Vector Concatenate(nonvirtual buses)	OK	OK
Zero-Order Hold	OK	OK

3.10.2 QGen conversion for bus data types

- Virtual homogeneous buses are converted to arrays with base type equal to that of bus leaf elements and the length corresponding to the total number of bus elements. In case of a nested bus, the elements are flattened in depth-first order.
- Virtual non-homogeneous buses are in general split into n variables with n being the total number of bus elements.
  - The one exception occurs when the data type of the virtual bus is explicitly defined in the associated Inport or Outport block, in which case the generated function parameters are structures of the given bus type.
- Non-virtual buses use the data type defined in Matlab workspace.

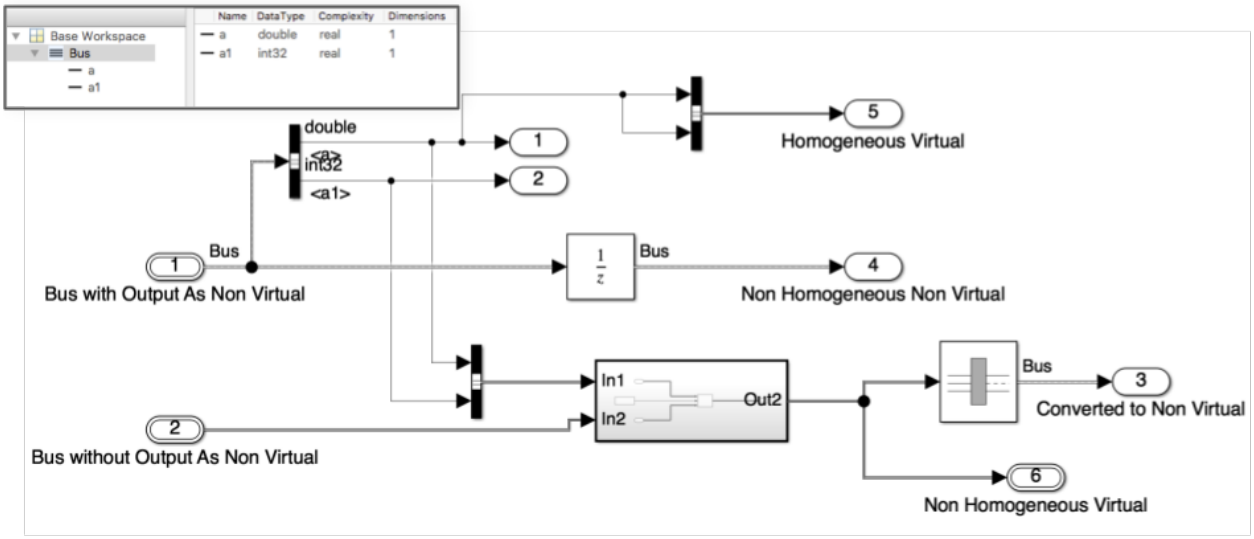
3.11 Custom Data Types

QGen supports custom type definitions based on `Simulink.Aliastype`, `Simulink.Bus` and `Simulink.IntEnumType` object definitions. For signals, parameters, variables etc. in the Simulink / Stateflow model that have been defined to be of such custom types QGen shall use the corresponding type. By default, QGen shall generate respective type definitions. However, it is also possible to bind those Simulink objects with external type definitions in the target language (Ada or C). Then QGen shall use the respective external definitions in the generated code. Generally, this facilitates the integration of generated and external application code. See *Custom Data Types Based on External Definitions* for more.

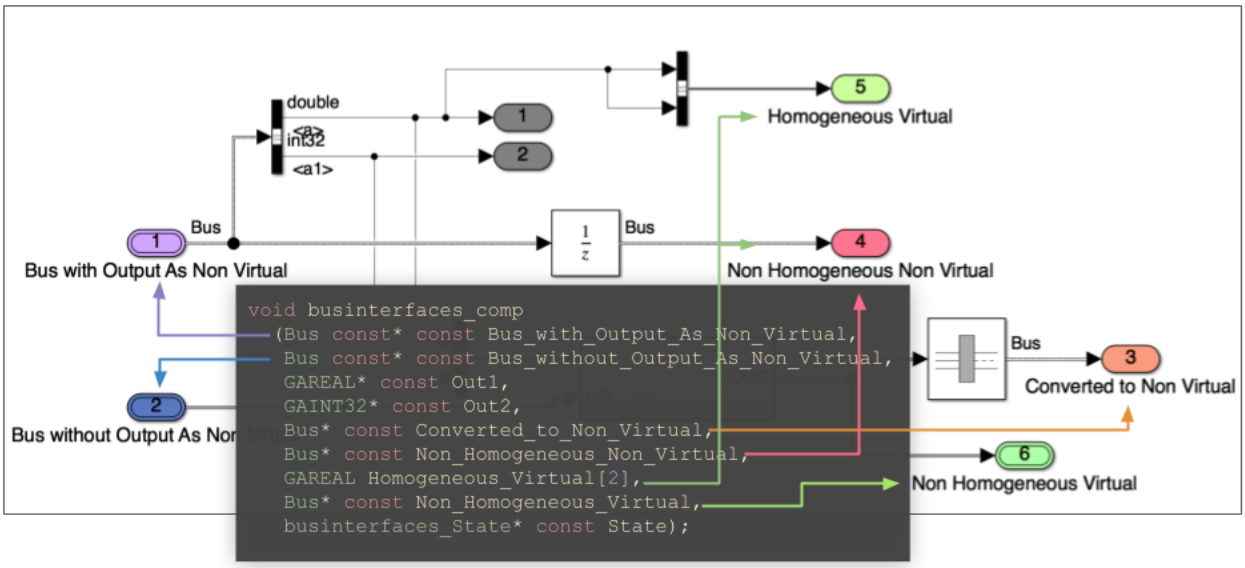
Note: The custom type definitions must be loaded into the MATLAB workspace before running QGen (before the XMI export phase). The typing information required for code generation will then be automatically extracted into the XMI files, as needed.

Note: Structured parameters don't require explicit type definitions and can also be given in a parameter file. See *MATLAB Code for Parameters in Workspace Containers and Block Parameters*.

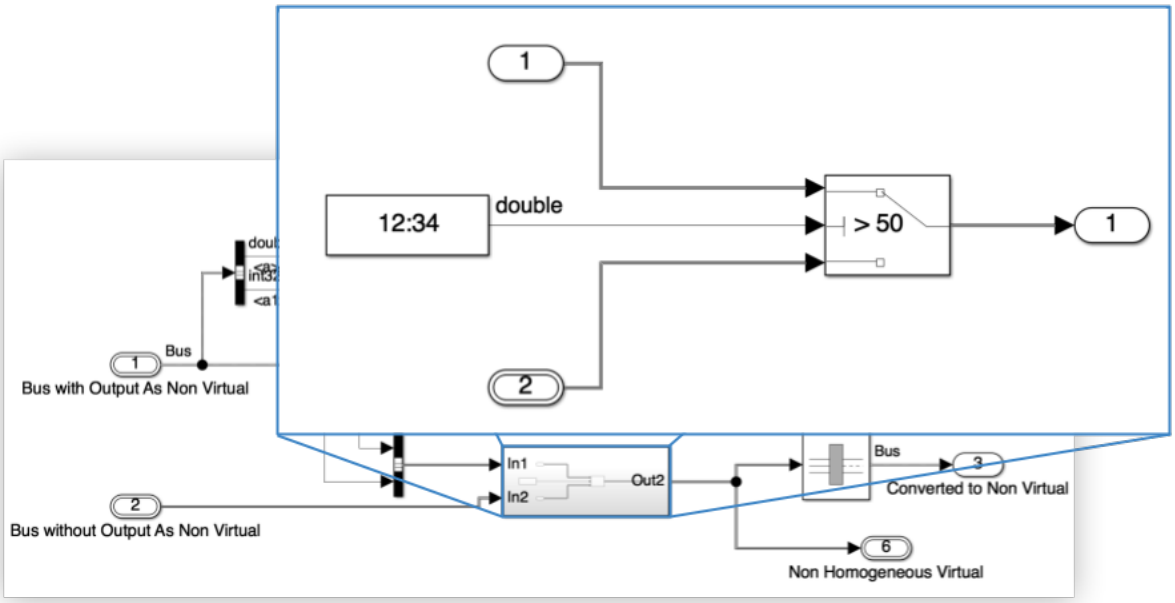
QGen will generate structures differently depending on the type of bus used. Considering the model below, that uses all the types of buses that QGen supports:



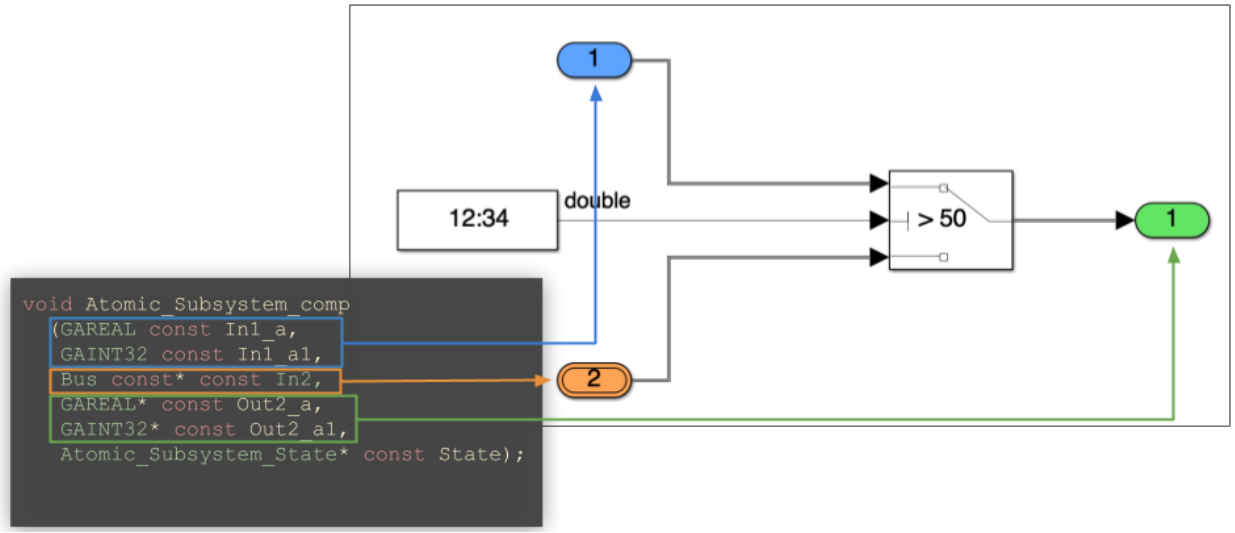
QGen will generate the following types (example is in C, but the equivalent Ada types would be used when generating Ada) :



- Virtual homogeneous buses are converted to arrays with base type equal to that of bus leaf elements and the length corresponding to the total number of bus elements. In case of a nested bus, the elements are flattened in depth-first order.
- Non-virtual buses use the datatype defined in the MATLAB workspace.



- Virtual non-homogeneous buses are in general split into n variables with n being the total number of bus elements. The one exception occurs when the datatype of the virtual bus is explicitly defined in the associated Inport or Outport block, in which case the generated function parameters are structures of the given bus type.



3.11.1 General Constraints for Custom Data Types

Array and Structure Types

Simulink doesn't support defining custom array (vector, matrix, etc.) types. Array types are usually defined implicitly as needed based on the base type and size of elements in the model. However, custom arrays with fixed size can be defined within buses (records/structures) as bus elements.

Structures containing arrays and arrays containing structures are supported with a limitation that

- when a structure is included in an array, it must have only scalar members and

- when a structure member is an array, the elements of this array must not be structures.

Enumerated Types

Enumerated types are supported. The integer values that are associated with enumeration literals don't have to be ordered or consecutive. However, when the chosen target language is Ada they must be distinct within the given enumerated type. This is a requirement in the Ada language. Such type definitions should be refined and the models updated accordingly. C code generation does not have this constraint. Duplicate values will map to the same value also in the corresponding C code.

3.11.2 QGen Attributes

For using custom data types with QGen some attributes must be specified in the Description field of the respective Simulink data objects. The Description can also contain other text. Only the keyword patterns indicated below will be processed by QGen.

Note: In older versions of Simulink the Description field is not available for `BusElement` objects (it is available for `Bus` objects only). This limitation does not affect Simulink versions 8.5 (R2015a) and above. For earlier versions the support of the Description field for `BusElement` objects should be checked. If it is not available, then the usage of custom types with QGen is more limited.

Table 1: Description of the supported QGen attributes

Attribute	Description
QGen.AdaSpecFile	(mandatory for Ada generation) The name of the external file containing custom type definitions
QGen.RangeName	(mandatory when the range does not start from 1) The name of the range type used for the array indices of a custom array type (represented as a bus)
QGen.ValueRange	(optional) The allowed range of values for a numeric type
QGen.TypeName	(optional) If specified, the name to be used for the type

Example:

```
% Alias to a standard numeric type in Simulink
MyReal = Simulink.AliasType;
MyReal.BaseType = 'double';
MyReal.Description = ...
   ['This field contains description about this data type ', ...
    'as well as QGen-specific attributes. ', ...
    'The latter must be separated from the rest of text by whitespace. ', ...
    'Hence, there is a space character after this sentence. ', ...
    'QGen.AdaSpecFile:mytypes.ads QGen.ValueRange:[0.0, 1000.0] ', ...
    ' More text follows after whitespace.'];
```

3.11.3 Value Range Checks

For aliases of numeric data types one can optionally specify also the expected numeric range of the data with the `QGen.ValueRange` attribute. For Ada code QGen will generate a cast that enforces the values to be in this range and will raise a run-time exception, if this is not the case. For C code such casts are not generated. However, this information can also be used by the QGen Model Verifier that performs static analysis of the model and can reveal certain design issues also statically.

Example:

```
% Alias to a standard numeric type in Simulink
MyReal = Simulink.AliasType;
MyReal.BaseType = 'double';
MyReal.Description = 'QGen.ValueRange:[0.0, 1000.0]';
```

3.11.4 Custom Data Types Based on External Definitions

QGen can use custom externally defined data types in the generated code instead of generating default type definitions. The functionality is slightly different when generating Ada or C code. The external type definitions are assumed to be in the same programming language as the code generation target language.

In both cases the external data types must also be defined in Simulink using `Simulink.Aliastype`, `Simulink.Bus` or `Simulink.IntEnumType` objects. For signals, parameters, variables etc. in the Simulink / Stateflow model that have been defined to be of such custom types QGen shall use the corresponding target language types as well. For elements defined using Simulink predefined types QGen shall generate default type definitions.

QGen also generates the respective `with` and `use` clauses (Ada) and `#include` statements (C) for the external type definitions. It is the user's task to ensure that the external files are accessible at compile time.

External Data Types for Ada Code Generation

The names of the Ada types and reference to the external spec (.ads) file containing the definitions for the target Ada code should be indicated in the `Description` field of Simulink data objects as shown below using the `QGen.AdaSpecFile` attribute.

Definition from external spec files may use standard Ada naming convention for packages with a hyphen as package delimiter, such as: `mytypes.ads`, `mytypes-subpackage.ads`, `mytypes-subpackage-basictypes.ads`. Generated files will then include them as:

```

```

with mytypes; use mytypes; with mytypes.subpackage; use mytypes.subpackage; with mytypes.subpackage.basictypes; use mytypes.subpackage.basictypes;

Simulink doesn't support defining custom array types. However, for the custom array types used within buses (records/structures) one can optionally also provide the name of the range type used for the array indices with the QGen attribute `QGen.RangeName`. Providing the range type is mandatory, when the range does not start from 1.

Examples

This MATLAB script demonstrates how to set up custom Ada types for using with Simulink and Stateflow. This example assumes that the external type definitions are in the file `mytypes.ads`. The content of the latter is provided further below.

Listing 3.1: MATLAB code for defining external Ada-based custom data types

```
% Alias to a standard numeric type in Simulink
MyReal = Simulink.AliasType;
MyReal.BaseType = 'double';
MyReal.Description = ...
   ['This field contains description about this data type ', ...
    'as well as QGen-specific attributes. ', ...
    'The latter must be separated from the rest of text by whitespace. ', ...
    'Hence, there is a space character after this sentence. ', ...
    'QGen.AdaSpecFile:mytypes.ads QGen.ValueRange:[0.0, 1000.0] ', ...
    ' More text follows after whitespace.'];

% Alias to a standard numeric type in Simulink
MyInt16 = Simulink.AliasType;
MyInt16.BaseType = 'int16';
MyInt16.Description = 'QGen.AdaSpecFile:mytypes.ads';

% Alias to a standard numeric type in Simulink
MyBool = Simulink.AliasType;
MyBool.BaseType = 'boolean';
MyBool.Description = 'QGen.AdaSpecFile:mytypes.ads';

% EnumType (defined in this script)
Simulink.defineIntEnumType('MyColors', ...
    {'UNDEFINED', 'RED', 'GREEN', 'BLUE'}, ...
    [0;1;2;3], ...
    'DefaultValue', 'UNDEFINED', ...
    'DataScope', 'Imported', ...
    'AddClassNameToEnumNames', false, ...
    'Description', 'QGen.AdaSpecFile:mytypes.ads Some more description ...');

% Bus/Struct Element data for MyStruct. 2 elements of numeric type MyReal.
MyStruct_member1=Simulink.BusElement;
MyStruct_member1.Name='arr1';
MyStruct_member1.DataType='MyReal';
MyStruct_member1.Dimensions=2;
MyStruct_member1.Description = ...
    ['QGen.AdaSpecFile:mytypes.ads QGen.TypeName:MyReal_2_array ', ...
    ' QGen.RangeName:MyReal_2_index'];

% Bus/Struct Element data for MyStruct. 3 elements of numeric type MyInt16.
MyStruct_member2=Simulink.BusElement;
MyStruct_member2.Name='arr2';
MyStruct_member2.DataType='MyInt16';
MyStruct_member2.Dimensions=3;
MyStruct_member2.Description = ...
    ['QGen.AdaSpecFile:mytypes.ads QGen.TypeName:MyInt16_3_array ', ...
    ' QGen.RangeName:MyInt16_3_index'];
```

```
% Bus/Struct Element data for MyStruct. 1 element of enumerated type MyColors.
MyStruct_member3=Simulink.BusElement;
MyStruct_member3.Name='enum1';
MyStruct_member3.DataType='Enum: MyColors';
MyStruct_member3.Dimensions=1;
MyStruct_member3.Description = ...
    ['QGen.AdaSpecFile:mytypes.ads QGen.TypeName:MyColors'];

% Bus/Struct Element data for MyStruct. 2 elements of enumerated type MyColors
    .
MyStruct_member4=Simulink.BusElement;
MyStruct_member4.Name='enum1Arr';
MyStruct_member4.DataType='Enum: MyColors';
MyStruct_member4.Dimensions=2;
MyStruct_member4.Description = ...
    ['QGen.AdaSpecFile:mytypes.ads QGen.TypeName:MyColors2_array ', ...
    ' QGen.RangeName:MyColors2_index'];

% Bus/Struct Element data for MyStruct. 1 element of enumerated type
% OtherColors, which has been defined in a dedicated .m file.
MyStruct_member5=Simulink.BusElement;
MyStruct_member5.Name='enum2';
MyStruct_member5.DataType='Enum: OtherColors';
MyStruct_member5.Dimensions=1;
MyStruct_member5.Description = ...
    ['QGen.AdaSpecFile:mytypes.ads QGen.TypeName:OtherColors'];

% Bus/Struct Element data for MyStruct. 2 elements of enumerated type
% OtherColors, which has been defined in a dedicated .m file.
MyStruct_member6=Simulink.BusElement;
MyStruct_member6.Name='enum2Arr';
MyStruct_member6.DataType='Enum: OtherColors';
MyStruct_member6.Dimensions=2;
MyStruct_member6.Description = ...
    ['QGen.AdaSpecFile:mytypes.ads QGen.TypeName:OtherColors2_array ', ...
    ' QGen.RangeName:OtherColors2_index'];

% Bus/Struct Type that uses the previous type definitions.
MyStruct=Simulink.Bus;
MyStruct.Elements = ...
    [MyStruct_member1, MyStruct_member2, MyStruct_member3, ...
     MyStruct_member4, MyStruct_member5, MyStruct_member6];
MyStruct.Description = ...
    ['QGen.AdaSpecFile:mytypes.ads QGen.TypeName:MyStruct'];
```

The file `mytypes.ads` has the following contents.

Listing 3.2: Ada type definitions to be used in the target code.

```
package mytypes
is

   --  MyReal  --
   subtype MyReal is Standard.Long_Float;
```

```
   -- MyReal_2_array --
   subtype MyReal_2_index is Integer range 1 .. 2;
   type MyReal_2_array is array (MyReal_2_index) of MyReal;

   -- MyInt16 --
   subtype MyInt16 is Integer range -32768 .. 32767;

   -- MyInt16_3_array --
   subtype MyInt16_3_index is Integer range 1 .. 3;
   type MyInt16_3_array is array (MyInt16_3_index) of MyInt16;

   -- MyBool --
   subtype MyBool is Boolean;

   -- MyColors --
   type MyColors is (UNDEFINED, RED, GREEN, BLUE);
   for MyColors use (UNDEFINED => 0, RED => 1, GREEN => 2, BLUE => 3);

   -- MyColors_array --
   subtype MyColors2_index is Integer range 1 .. 2;
   type MyColors2_array is array (MyColors2_index) of MyColors;

   -- OtherColors --
   type OtherColors is (CYAN, MAGENTA);
   for OtherColors use (CYAN => 5, MAGENTA => 7);

   -- OtherColors_array --
   subtype OtherColors2_index is Integer range 1 .. 2;
   type OtherColors2_array is array (OtherColors2_index) of OtherColors;

   -- MyStruct --
   type MyStruct is record
      arr1     : MyReal_2_array;
      arr2     : MyInt16_3_array;
      enum1    : MyColors;
      enum1Arr : MyColors2_array;
      enum2    : OtherColors;
      enum2Arr : OtherColors2_array;
   end record;

end mytypes;
```

External Data Types for C Code Generation

Reference to the external header (.h) file containing the type definitions for the target C code should be indicated in the `HeaderFile` field of Simulink data objects as shown below.

Configuring the Custom Code Section for Simulation

Note: If the model contains also Stateflow charts, then before simulating you must also configure Simulink to read the data from this header file. For this you should open the Configuration Parameters dialog box Simulation Target pane and enter the custom header file and directory as needed. See also the corresponding figure.

Tip: The data in these fields is only used by Simulink and Stateflow. QGen does not read them. Also, one should separately make sure that the file is available also when compiling and linking the generated code.

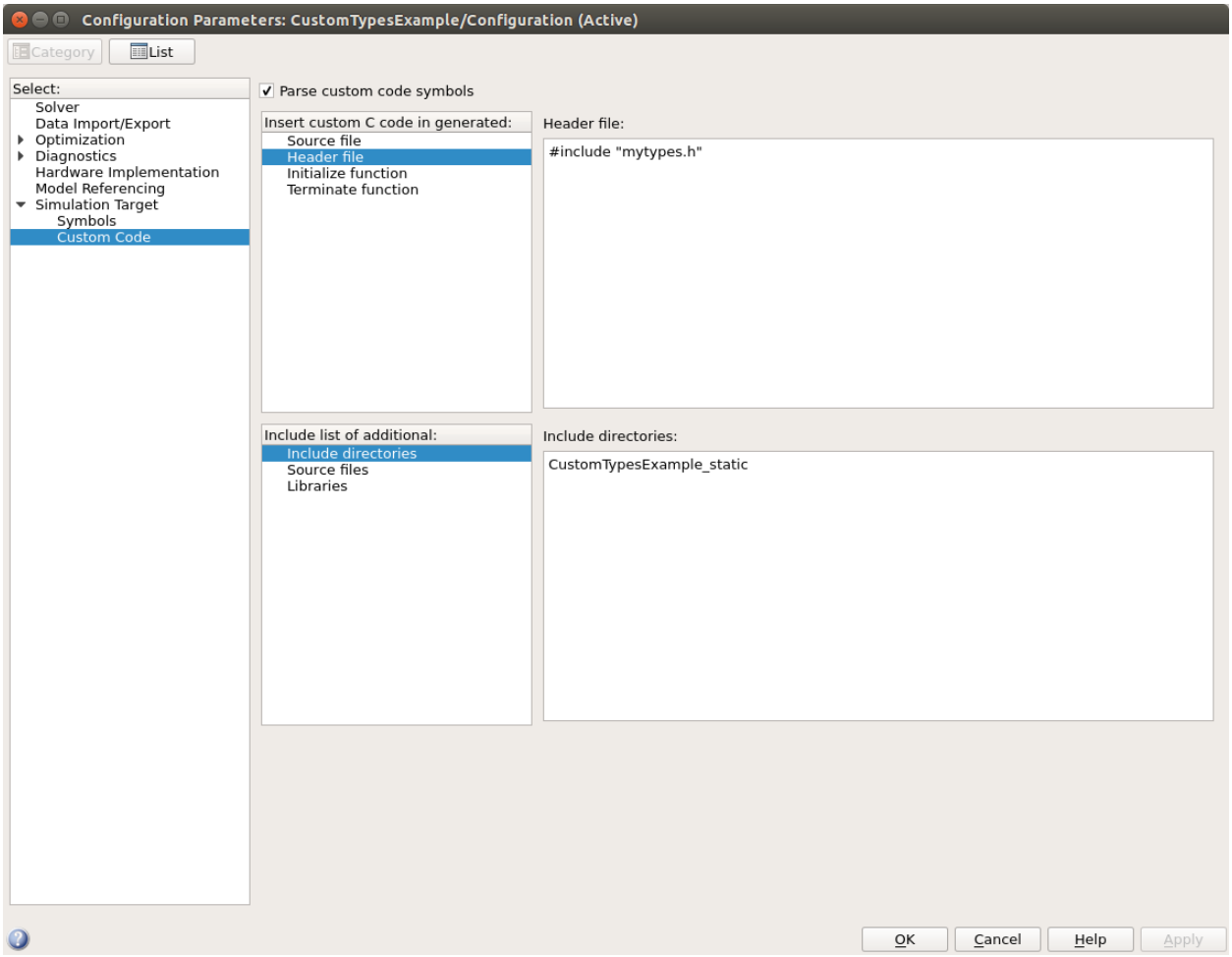


Fig. 1: Configuring Simulink to use a custom C header file for simulation.

Examples

The MATLAB script below demonstrates how to set up custom C types for using with Simulink and Stateflow. This example assumes that the external type definitions are in the file `mytypes.h`. The content of the latter is provided further below.

Listing 3.3: MATLAB code for defining external C-based custom data types.

```
% Alias to a standard numeric type in Simulink
MyReal = Simulink.AliasType;
MyReal.BaseType = 'double';
MyReal.HeaderFile = 'mytypes.h';
MyReal.Description = ...
   ['This field contains description about this data type ', ...
    'as well as QGen-specific attributes. ', ...
    'The latter must be separated from the rest of text by whitespace. ', ...
    'Hence, there is a space character after this sentence. ', ...
    'QGen.ValueRange:[0.0, 1000.0] ', ...
    ' More text follows after whitespace.'];

% Alias to a standard numeric type in Simulink
MyInt16 = Simulink.AliasType;
MyInt16.BaseType = 'int16';
MyInt16.HeaderFile = 'mytypes.h';

% Alias to a standard numeric type in Simulink
MyBool = Simulink.AliasType;
MyBool.BaseType = 'boolean';
MyBool.HeaderFile = 'mytypes.h';

% EnumType (defined in this script)
Simulink.defineIntEnumType('MyColors', ...
    {'UNDEFINED', 'RED', 'GREEN', 'BLUE'}, ...
    [0;1;2;3], ...
    'DefaultValue', 'UNDEFINED', ...
    'DataScope', 'Imported', ...
    'AddClassNameToEnumNames', false, ...
    'HeaderFile', 'mytypes.h', ...
    'Description', 'Some description ...');

% Bus/Struct Element data for MyStruct. 2 elements of numeric type MyReal.
MyStruct_member1=Simulink.BusElement;
MyStruct_member1.Name='arr1';
MyStruct_member1.DataType='MyReal';
MyStruct_member1.Dimensions=2;

% Bus/Struct Element data for MyStruct. 3 elements of numeric type MyInt16.
MyStruct_member2=Simulink.BusElement;
MyStruct_member2.Name='arr2';
MyStruct_member2.DataType='MyInt16';
MyStruct_member2.Dimensions=3;

% Bus/Struct Element data for MyStruct. 1 element of enumerated type MyColors.
MyStruct_member3=Simulink.BusElement;
MyStruct_member3.Name='enum1';
```

```
MyStruct_member3.DataType='Enum: MyColors';
MyStruct_member3.Dimensions=1;

% Bus/Struct Element data for MyStruct. 2 elements of enumerated type MyColors
    .
MyStruct_member4=Simulink.BusElement;
MyStruct_member4.Name='enum1Arr';
MyStruct_member4.DataType='Enum: MyColors';
MyStruct_member4.Dimensions=2;

% Bus/Struct Element data for MyStruct. 1 element of enumerated type
% OtherColors, which has been defined in a dedicated .m file.
MyStruct_member5=Simulink.BusElement;
MyStruct_member5.Name='enum2';
MyStruct_member5.DataType='Enum: OtherColors';
MyStruct_member5.Dimensions=1;

% Bus/Struct Element data for MyStruct. 2 elements of enumerated type
% OtherColors, which has been defined in a dedicated .m file.
MyStruct_member6=Simulink.BusElement;
MyStruct_member6.Name='enum2Arr';
MyStruct_member6.DataType='Enum: OtherColors';
MyStruct_member6.Dimensions=2;

% Bus/Struct Type that uses the previous type definitions.
MyStruct=Simulink.Bus;
MyStruct.Elements = ...
    [MyStruct_member1, MyStruct_member2, MyStruct_member3, ...
     MyStruct_member4, MyStruct_member5, MyStruct_member6];
MyStruct.HeaderFile = 'mytypes.h';
```

The file `mytypes.h` has the following contents.

Listing 3.4: C type definitions to be used in the target code.

```
#ifndef __MYTYPES__
#define __MYTYPES__

typedef double MyReal;

typedef short MyInt16;

typedef unsigned char MyBool;

typedef enum {
   UNDEFINED = 0,
   RED = 1,
   GREEN = 2,
   BLUE = 3
} MyColors;

typedef enum {
   CYAN = 5,
   MAGENTA = 7
} OtherColors;
```

```
typedef struct {
    MyReal arr1[2];
    MyInt16 arr2[3];
    MyColors enum1;
    MyColors enum1Arr[2];
    OtherColors enum2;
    OtherColors enum2Arr[2];
} MyStruct;

#endif
```

3.12 Naming Convention

When generating code, QGen uses the names of model elements as much as possible. The elements that explicitly reference external code or another model are expected to have a unique name that is always preserved during code generation. Such elements are referred as **fixed identifiers**.

Table 2: **Generated identifiers with unchanged model element names**

Model element	Code element
Data type	Data type
Workspace variable	Module-level variable
Model name	Module name

Other elements have a generated name which is derived from corresponding model element name but may be decorated to end up with a unique identifier. The table below lists named elements from the input model and corresponding rules for using their name in generated code. We refer to these identifiers as **derived identifiers**

Table 3: **Generated identifiers derived from model element names**

Model element	Code element
Subsystem name (atomic subsystem)	Module name
	Prefix of initialization (`_initStates`, `_initOutputs`), compute (`_comp`) and memory update (`_up`), enable (`_enable`), disable (`_disable`) functions
Subsystem port (atomic subsystem)	Argument of initialization (`_initStates`, `_initOutputs`), compute (`_comp`) and memory update (`_up`), enable (`_enable`), disable (`_disable`) functions (the default mode)
	Structure element of IO structure when generating with `--wrap-io` switch
	Suffix of global variable corresponding to a port when generating with `--global-io` switch
Block name	Prefix of output variables generated from this block unless the outgoing signal has a name
	Prefix of memory variables generated from this block
Port name	Suffix of output variables generated from this block unless the outgoing signal has a name
Signal name	Name of the block output variable corresponding to this signal. In case there are several named signals starting from the same port, QGen uses the first name it finds.

In the third group, referred as **generated identifiers** there are identifiers that are created during code generation process and do not correspond directly to a single model element.

Table 4: **Generated identifiers with no corresponding name in the model**

Code element	Naming rule
Array data type (Ada only)	`<type name>_Array_<dimension1 length>[_<dimension2 length>]`
Array range (Ada only)	`<array type name>_Range_<dimension number>`

Identifiers derived from model element names are first cleaned of illegal characters (such as whitespace, umlauts etc). The illegal characters are replaced by underscores, two subsequent underscores are merged to one. In case an identifier starts with a number, it is prefixed with "z" in C or "u" in Ada.

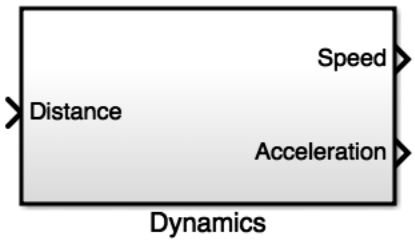
Finally the overlapping names are checked for uniqueness. In case of conflicts with each other the name is suffixed with underscore and a number. When generated name overlaps with a reserved word it is prefixed with "z" in C or duplicated first letter in Ada.

The rules for modifying the generated identifiers are as follows:

- the names of fixed identifiers and generated identifiers are not modified during the code generation process
- names of functions and types linked to a library instance are suffixed by the first 13 characters of the checksum associated to that library.
- illegal characters in fixed names are not checked and shall be discovered by the target compiler
- when there is a naming conflict between two fixed identifiers or between a fixed identifier and generated identifier the code generation stops with error
- name conflict between two generated identifiers can not occur — in case the generated element (e.g. data type) gets a name that already exists, the existing element is used instead.
- generated names are mangled as described above. In case of a conflict between two generated names the one discovered later by code generator is usually mangled except in case one of them is function argument. In case of a conflict between function argument an identifier inside of that function the latter is modified.

3.12.1 Examples

In a model like the following:



the name of the atomic subsystem `Dynamics` determines the name of the compute function (`Dynamics_comp`), and the names of the input and output ports are exactly the same as the name of the parameters:

```
void Dynamics_comp
  (GAREAL const Distance,
   GAREAL* const Speed,
   GAREAL* const Acceleration)
```

Note that when using the `--global-io` switch to generate main subsystem's IO as global variables, the names of the variables for the previous model contain both the names of the subsystem and the ports:

```
GAREAL Dynamics_comp_Distance;
GAREAL Dynamics_comp_Speed;
GAREAL Dynamics_comp_Acceleration;
```

If subsystem Dynamics contained signals named

- `Distance`
- `else`
- `1_speed`
- `name_____with_ underscores`

then the generated variable names would be:

- `Distance_1` — there was a function argument named `Distance`
- `zelse` for C or `eelse` for Ada — "else" is a reserved keyword
- `z1_speed` for C or `u1_speed` for Ada — derived from `1_speed`. The identifier cannot start with a number
- `name_with_underscores` — space is replaced with underscore, subsequent underscores are merged

3.13 Customizing Arithmetic Functions

Thanks to the `--arith-conf` switch, it is possible to customize the call for arithmetic functions on both scalar and matrices. The switch take in input a textual file containing the specification of the transformation in the following format:

```
TRANS ::= ARITY : SATURATION : SIGNATURE : MODULE
```

where the arity of the arithmetic operation expressed as:

```
ARITY ::= TYPE([]([])?)? = TYPE([]([])?)? OPERATOR TYPE([]([])?)?

TYPE  ::= double|single|int32|int16|int8|uint32|uint16|uint8|boolean

OPERATOR ::= +|-|*|/|#
```

where the # operator is the element-wise multiplication for matrices.

SATURATION is a boolean value 'true' or 'false', where 'true' indicates that the custom call replaces a saturated operator.

SIGNATURE is the signature of the function to be called, using the special MATLAB variables uN, yN and the size function.

Finally, MODULE is the name of the .h/.ads file where the external functions are located.

For example, the following text:

```
double = double+double : false : double y1 = my_add(double u1, double u2) :
my_lib.h
```

means that, whenever the '+' operator between double is found, it should be replaced by a call to function `my_add` declared in file `my_lib.h`. In this case, `u1` represents the first operand, while `u2` represents the second. With the transformation above, the code:

```
double a, b, c;
c = a + b;
```

is transformed into:

```
#include "my_lib.h"
...
double a, b, c;
c = my_add (a, b);
```

When it comes to matrices, it is typical to call a void function, for example:

```
double[][] = double[][]+double[][] : false : my_element_wise_sum
                (double u1[][], double u2[][],
                int32 size(u2, 1), int32 size(u2, 2)) : my_lib.h
```

In this case, the matrix multiplication is replaced by a call to the my_matrix_element_wise function. Since the function signature does not contain a return value, an implicit parameters is added ad the end of the call. With the transformation above, the code:

```
double[][] a, b, c;
...
for (i = 0; i < size_1; i++){
    for (j = 0; j < size_2; j++){
      c[i][j] = a[i][j] + b[i][j];
    }
}
```

is transformed into:

```
#include "my_lib.h"
...
double[][] a, b, c;
my_element_wise_sum (a, b, size_1, size_2, c);
```

Since C does not have a proper syntax to pass two dimensional arrays as arguments, matrices are passed as serialized row-major vectors:

```
void my_element_wise_sum (double a[], double b[], int size_1, int size_2,
                          double c[])
{
  int i, j;
  for (i = 0; i < size_1; ++i)
  {
      for (j = 0; j < size_2; ++j)
        c[i * size_2 + j] = ...; /* instead of c[i][j] */
  }
}
```

An example of a configuration file is available in `share/qgen/examples/customization/arith`.

3.14 Customizing Block Implementations

With the `--block-conf` switch, it is possible to replace the implementation of supported blocks with functions calls. The switch takes as parameter a textual file containing the specifications for block replacement:

```
TRANS ::= BLOCK_TYPE : ARITY : (PARAMETER_FILTERS)?
                     : (INIT_SIGNATURE)? : (COMPUTE_SIGNATURE)?
                     : (UPDATE_SIGNATURE)? : MODULE
```

Where the arity of the block is expressed as:

```
ARITY ::= TYPE([]([])?)?(,TYPE([]([])?)?)* <- TYPE([]([])?)?(,TYPE([]([])?)?)*

TYPE  ::= double|single|int32|int16|int8|uint32|uint16|uint8|boolean
```

Where the types on the left of '<-' represent the types of the output ports of the block and the types on the right are the input ports of the block.

PARAMETER_FILTERS is a list of parameter names associated to a value:

```
PARAMETER_FILTERS ::= (PARAM_NAME => VALUE (# PARAM_NAME => VALUE)*)?
```

INIT_SIGNATURE is the signature of the function to call for the initialization step of the block.

COMPUTE_SIGNATURE is the signature of the function called at the computation steps of the block.

UPDATE_SIGNATURE is the signature of the function called after each computation steps to update internal data.

The three signatures use the MATLAB special variables uN, yN and the size function. The feature also provides special variables mN to designate persistent memory variables and allows references to block parameters by using the parameter's name.

Finally, MODULE is the name of the .h/.ads file where the external functions are located.

For example, the following line:

```
Gain : double <- double : : : double y1 = gain_comp(double u1) : : libgain.h
```

tells that all Gain blocks whose input and output are of type double should be replaced by a call to function gain_comp declared in file libgain.h. No init update function are needed here since there is no persistent memory at stake.

Another example with parameter filtering:

```
Abs : double<-double : SaturateOnIntegerOverflow => off : \
    : double y1 = abs_comp(double u1) : : libabs.h
```

This tells to replace only Abs blocks whose parameter SaturateOnIntegerOverflow is off by a call to abs_comp. As is pattern matching, you can define a "default" case for block of the same type and arity but didn't match the parameter filters:

```
Abs : double<-double : : : double y1 = abs_sat_comp(double u1) : : libabs.h
```

This line in combination with the one above will make QGen replace Abs blocks using saturation.

Here's another example with using memory variables an parameter references:

```
UnitDelay : double[]<-double[] : \
          : ud_init (double InitialValue[], uint32 size(InitialValue,1), \
                     double m1[size(InitialValue,1)], uint32 size(m1,1)) \
          : ud_comp (double y1[], uint32 size(y1,1), double m1[], \
                     uint32 size(m1,1)) \
          : ud_up (double u1[], uint32 size(u1,1), double m1[], \
                   uint32 size(m1,1)) \
          : libunitd.h
```

This line provides the three function signatures to replace UnitDelay blocks:

- ud_init: This function is called for initialization, "double m1[size(InitialValue,1)]" will at the same time define a new memory persistent variable and specify that the created variable shall be passed to the function. The lengths of the arrays can also be provided as arguments to the function with the size() function (often needed when interfacing with C code).

  Note: When variables mN are of type array, the size of each dimension **must** be provided in the INIT_SIGNATURE. The numbering of mN variables shall start from one and the variables should appear in INIT_SIGNATURE in increasing order according to their numbering (no index shall be omitted).

- ud_comp: For computation, the unit delay needs to set its output y1 with its memory variable m1. Since the variable is already allocated, there is no need to provide its size. However, as in the above case, the function might still need to have the lengths of arrays as arguments.
- ud_up: Finally, the update function updates the m1 variable with the input of the unit delay block.

An example of configuration file is available in' *share/qgen/examples/customization/blocks*.

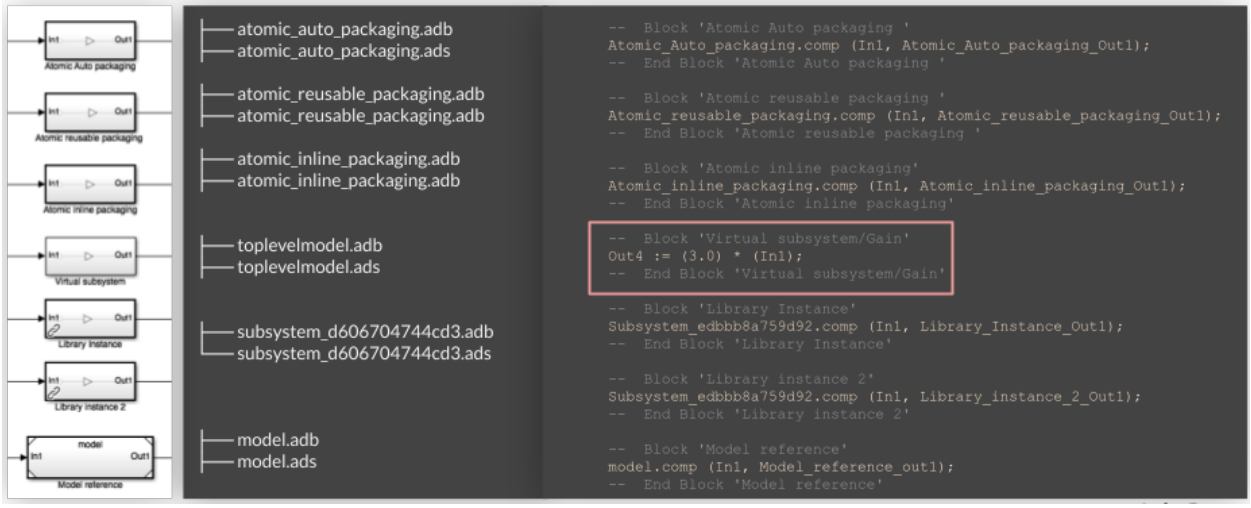
Generating code with the `--blocks-signatures` on a given model switch will generate a file called "qgen_<model_name>_blocks_signatures.txt. This file will contain the signatures of all blocks present in the model. These signatures can be used as a basis to create custom configuration files.

3.15 Detailed Code Generation Options

3.15.1 Flattening

The following options are available and influence the generated code packaging, in the form of Ada package or C body/header pair.

- Flatten non reusable subsystems: Atomic subsystems do not create their own package if they are not marked as reusable. Use Simulink Function Packaging in the Code Generation options of the Subsystem to specify that property.
- Flatten all: Specifies to flatten all subsystems and models in a single package.
- Flatten references: Default. Each model is flatten in its own package, but library instances are still factorized.
- No flattening: Maximum amount of packages created. Only virtual subsystems do not end up in their own package.



3.15.2 Coding Rules and Formatting

Remove unused enable

Off by default. When set to on, unused code generated for enable blocks will be removed from the generated code. This has the downside of no longer making the code generation modular and disables the capability to generate code incrementally from these sources.

Consts as vars

Off by default. QGen will generate constants as variables allowing the generated code to be tuned at runtime.

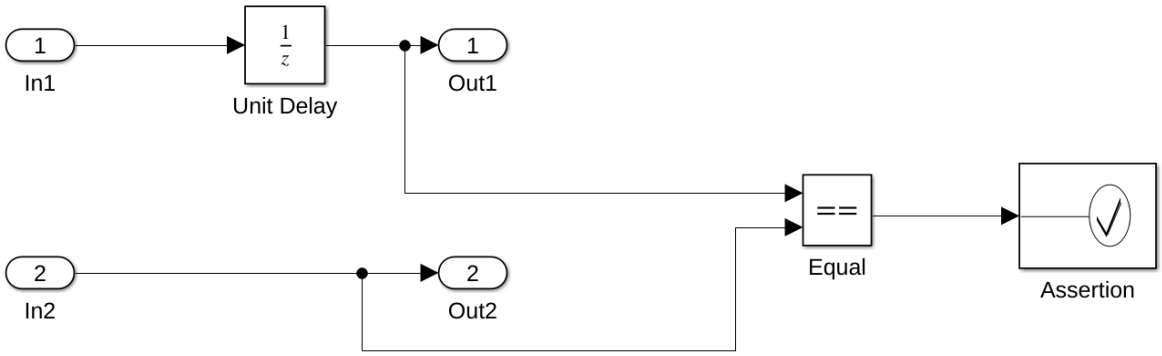
Remove goto

Off by default. Removes goto statement, only relevant for Stateflow Charts code.

Remove assertions

Remove code for Assertion blocks and their inputs.

Example:



Without the "Remove assertion" option:

```
void example_remove_assertion_comp
 (GAREAL const In2,
  GAREAL* const Out1,
  GAREAL* const Out2,
  example_remove_assertion_State* const State)
{
  /* Block 'example_remove_assertion/Unit Delay' */
  /* Block 'example_remove_assertion/Out1' */
  *Out1 = State->Unit_Delay_memory;
  /* End Block 'example_remove_assertion/Out1' */
  /* End Block 'example_remove_assertion/Unit Delay' */

  /* Block 'example_remove_assertion/In2' */
  /* Block 'example_remove_assertion/Out2' */
  *Out2 = In2;
  /* End Block 'example_remove_assertion/Out2' */
  /* End Block 'example_remove_assertion/In2' */

  /* Block 'example_remove_assertion/Equal' */
  /* Block 'example_remove_assertion/In2' */
  /* Block 'example_remove_assertion/Unit Delay' */
  /* Block 'example_remove_assertion/Assertion' */
  assert (State->Unit_Delay_memory == In2);
  /* End Block 'example_remove_assertion/Assertion' */
  /* End Block 'example_remove_assertion/Unit Delay' */
  /* End Block 'example_remove_assertion/In2' */
  /* End Block 'example_remove_assertion/Equal' */

}
```

With the "Remove assertion" option:

```
void example_remove_assertion_comp
 (GAREAL const In2,
```

(continues on next page)

(continued from previous page)

```
  GAREAL* const Out1,
  GAREAL* const Out2,
  example_remove_assertion_State* const State)
{
  /* Block 'example_remove_assertion/Unit Delay' */
  /* Block 'example_remove_assertion/Out1' */
  *Out1 = State->Unit_Delay_memory;
  /* End Block 'example_remove_assertion/Out1' */
  /* End Block 'example_remove_assertion/Unit Delay' */

  /* Block 'example_remove_assertion/In2' */
  /* Block 'example_remove_assertion/Out2' */
  *Out2 = In2;
  /* End Block 'example_remove_assertion/Out2' */
  /* End Block 'example_remove_assertion/In2' */

}
```

Unroll threshold

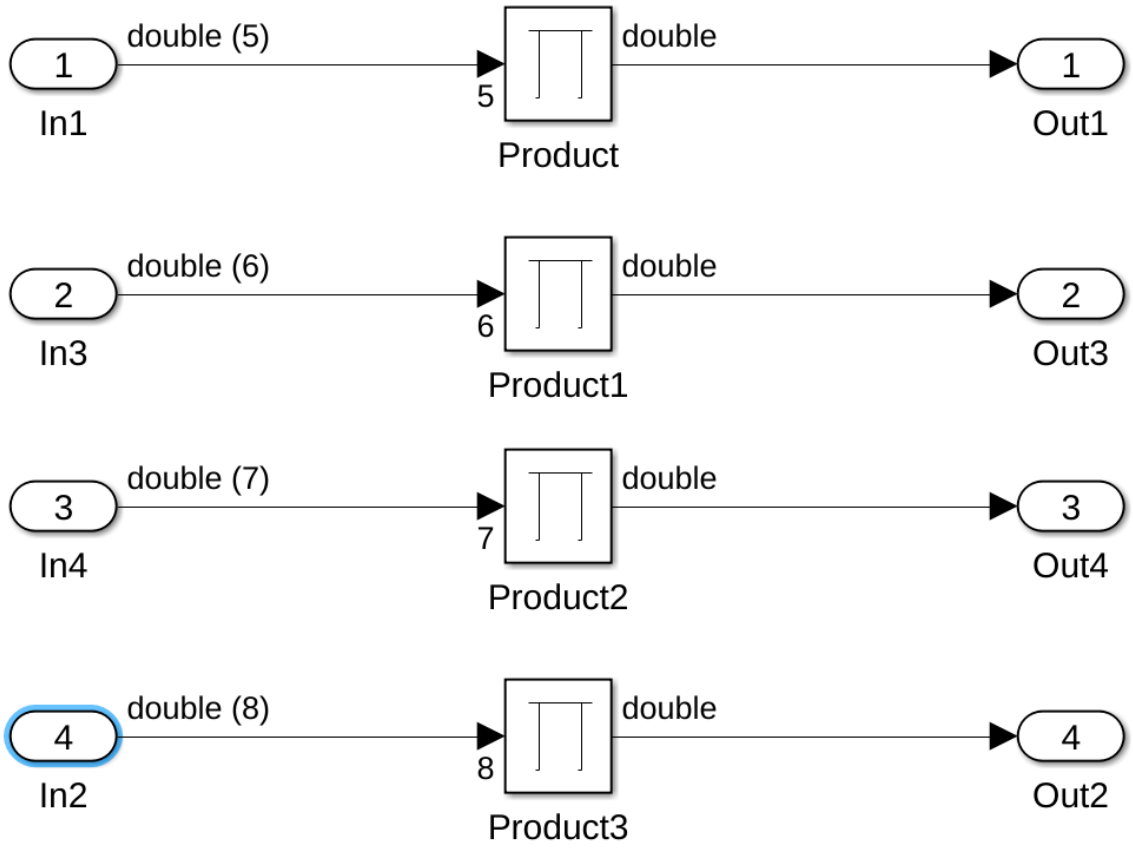
Modifies the way binary operations on arrays are generated.

When an array has less or the same number of elements than the specified threshold, binary operations on the array will be generated as a sequence of elementary binary operations on each element. Otherwise, the binary operations will be expanded to a loop statement iterating over the elements of the array. No value by default, in which case QGen uses 6 as the default value.

This code generation option mainly applies to blocks that perform arithmetic operations on a single, non-scalar input, e.g.:

- Product
- Sum
- Logic
- BitwiseOperator

Example with a threshold of 7, where only the Product3 block has a vector input with more than 7 elements:



As can be seen in the generated code (example for C), the product operations for blocks Product, Product1 and Product2 are generated as a sequence of product operations. However, for Product4, a loop statement iterating over all elements of the input is generated.

```
void unroll_threshold_example_comp
 (GAREAL const In1[5],
  GAREAL const In3[6],
  GAREAL const In4[7],
  GAREAL const In2[8],
  GAREAL* const Out1,
  GAREAL* const Out3,
  GAREAL* const Out4,
  GAREAL* const Out2)
{
  GAUINT8 i;

  /* Block 'unroll_threshold_example/Product' */
  /* Block 'unroll_threshold_example/In1' */
  /* Block 'unroll_threshold_example/Out1' */
  *Out1 = 1.0E+00 * In1[0] * In1[1] * In1[2] * In1[3] * In1[4];
  /* End Block 'unroll_threshold_example/Out1' */
  /* End Block 'unroll_threshold_example/In1' */
  /* End Block 'unroll_threshold_example/Product' */
```

(continues on next page)

(continued from previous page)

```
  /* Block 'unroll_threshold_example/Product1' */
  /* Block 'unroll_threshold_example/In3' */
  /* Block 'unroll_threshold_example/Out3' */
  *Out3 = 1.0E+00 * In3[0] * In3[1] * In3[2] * In3[3] * In3[4] * In3[5];
  /* End Block 'unroll_threshold_example/Out3' */
  /* End Block 'unroll_threshold_example/In3' */
  /* End Block 'unroll_threshold_example/Product1' */

  /* Block 'unroll_threshold_example/Product2' */
  /* Block 'unroll_threshold_example/In4' */
  /* Block 'unroll_threshold_example/Out4' */
  *Out4 = 1.0E+00 * In4[0] * In4[1] * In4[2] * In4[3] * In4[4] * In4[5] * In4[6];
  /* End Block 'unroll_threshold_example/Out4' */
  /* End Block 'unroll_threshold_example/In4' */
  /* End Block 'unroll_threshold_example/Product2' */

  /* Block 'unroll_threshold_example/Product3' */
  /* Block 'unroll_threshold_example/In2' */
  /* Block 'unroll_threshold_example/Out2' */
  *Out2 = 1.0E+00;
  for (i = 0; i <= 7; i++) {
     *Out2 = *Out2 * In2[i];
  }
  /* End Block 'unroll_threshold_example/Out2' */
  /* End Block 'unroll_threshold_example/In2' */
  /* End Block 'unroll_threshold_example/Product3' */

}
```

CHAPTER

FOUR

QGEN COMPATIBILITY CHECKER

QGen supports a subset of Simulink/Stateflow block types and features. The QGen Compatibility Checker allows you to check if your model contains elements that are incompatible with QGen and provides a report of violations and warnings so that you can investigate them. The report can be produced either as an HTML document (the default format) or a plain text document.

4.1 Check Levels

The QGen Compatibility Checker is invoked on a model or a subsystem in a model. It processes the elements of the model checking for incompatibilities with QGen and traverses referenced models and checks them too.

The QGen Compatibility Checker can be invoked with 3 levels of checks. The first one is the quickest but the least thorough. It is convenient for getting a quick initial assessment of your model(s). The last one is the slowest but the most thorough.

1. **Quick Check**

   This level verifies that the following properties are satisfied:

   - The solver type of the model and all referenced models is 'FixedStepDiscrete'. Violations are reported as failures.
   - The Simulink block types used in the model and in all referenced models and libraries are supported by QGen. This check is based on the *list of QGen-supported Simulink block types*. Violations are reported as failures, except for occurences of the 'MATLAB Function' block type which is reported as a warning since QGen handles this block by generating a skeleton to be implemented manually.

2. **Data Types and Callbacks Check**

   This check includes the *Quick Check*, and verifies additionally that the following properties are satisfied:

   - All port types do not have matrices with more than 2 dimensions. Violations are reported as failures.
   - There are no callback functions defined in the model and the contained blocks and ports. Violations are reported as warnings because it is up to the user to manually review their compatibility with QGen. Callback functions used for transfering mask parameter values from masks to parameters with the same name inside a masked subsystem are supported. Callback functions that modify the structure of the model by adding/removing blocks are not supported by QGen.

3. **Full Check**

   This check includes the *Quick Check* and the *Data Types and Callbacks Check*, and finally it invokes the initial steps of the code generation process to perform more thorough checks of the *Simulink block constraints*.

   For example, a *Delay* block that has the *Delay length* set to *Input port* will not be reported as incompatible by the *Quick Check*, but will be reported as unsupported by the *Full Check* in observance of the *Delay block constraints*.

4.2 Compatibility Report

The default format of the compatibility report is an HTML document including a list of the checks performed and the status of each check: Pass, Fail or Warning. The report includes collapsable sections to enhance readability.

Within the scope of each check, the result for each referenced model is included separately, with a listing of all references to that model from the main model to improve navigation.

Whenever possible, model elements are provided in the report as hyperlinks that navigate automatically to the corresponding location in the model when clicked.

An alternate plain text report format is available when invoking the checker from the MATLAB command line. However it is less detailed and less readable for large models. Therefore the recommended report format is the HTML document.

4.3 Invocation from the Simulink GUI

The QGen Compatibility Checker can be invoked from the QGen main menu, or by right-clicking a particular subsystem of your model.

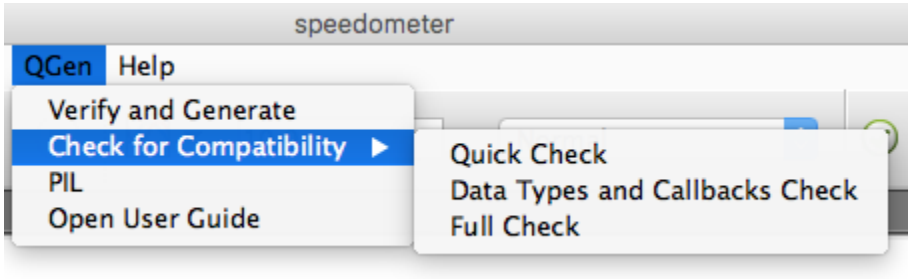


Fig. 1: QGen Compatibility Checker in the main menu

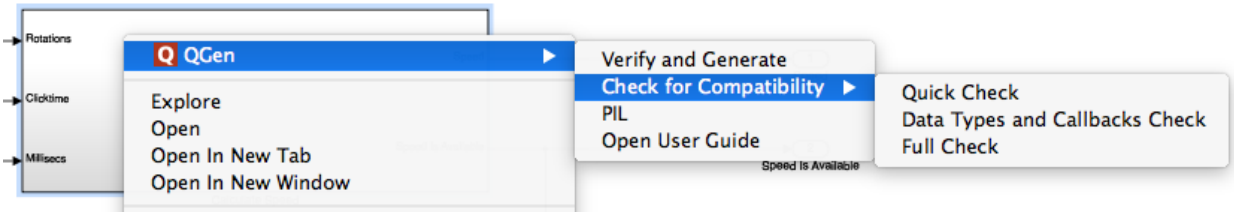


Fig. 2: QGen Compatibility Checker in the context menu

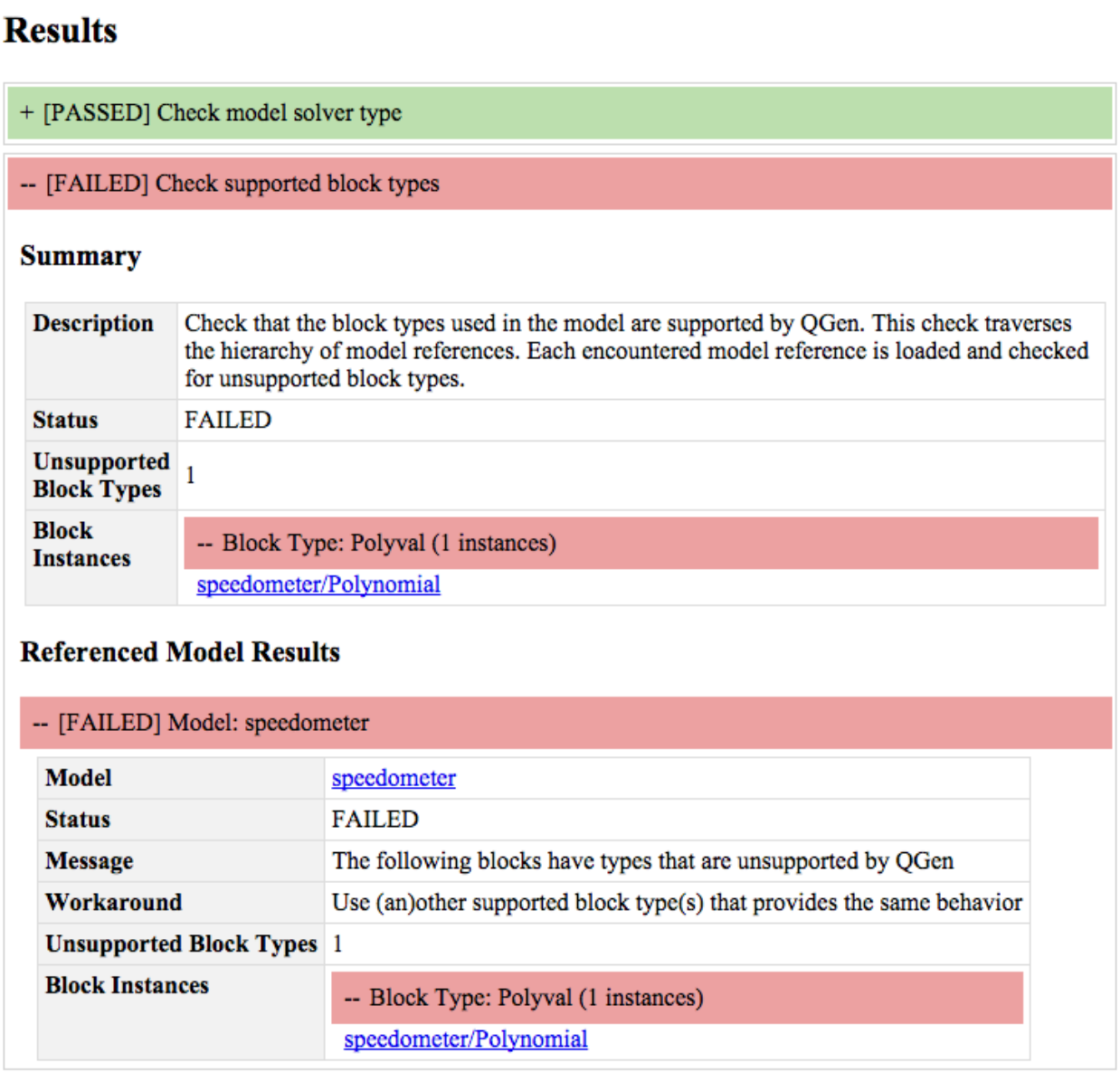
After selecting the desired check level, the Compatibility Checker will run the corresponding verifications and report its progress on the MATLAB command line. Progress is reported as `(Number of checked models / Total number of models)`.

Finally, the compatibility report is opened automatically in the MATLAB web browser.

Example

Let us consider the example model `share/qgen/examples/speedometer/speedometer.mdl` provided with the QGen installation. Let us introduce some incompatibilities such as a *Polynomial* block and a *Delay* block configured with *Delay length* set to *Input port*.

A *Quick Check* would only detect the unsupported *Polynomial* block and yield the following compatibility report:



Results

+ [PASSED] Check model solver type

-- [FAILED] Check supported block types

Summary

Description	Check that the block types used in the model are supported by QGen. This check traverses the hierarchy of model references. Each encountered model reference is loaded and checked for unsupported block types.
Status	FAILED
Unsupported Block Types	1
Block Instances	-- Block Type: Polyval (1 instances) speedometer/Polynomial

Referenced Model Results

-- [FAILED] Model: speedometer

Model	speedometer
Status	FAILED
Message	The following blocks have types that are unsupported by QGen
Workaround	Use (an)other supported block type(s) that provides the same behavior
Unsupported Block Types	1
Block Instances	-- Block Type: Polyval (1 instances) speedometer/Polynomial

Fig. 3: Compatibility Report with Quick Check level

However, a *Full Check* will also detect the unsupported *Delay* block because of its unsupported parameter value, and will yield the following compatibility report:

Results

+ [PASSED] Check model solver type

+ [FAILED] Check supported block types

+ [WARNING] Check callback functions

+ [PASSED] Check that the model compiles

+ [PASSED] Check supported data types

-- [FAILED] Fully check all constraints

Summary

Description	Check that port types, parameter types and their combinations are supported by QGen. This check is done by generating the decoration file and attempting the first steps of code generation which check detailed block-specific constraints and report any violations.
Status	FAILED

Referenced Model Results

-- [FAILED] Model: speedometer

Model	speedometer
Status	FAILED
Message	Result of the full compatibility check:
Elements (1)	speedometer
Details	[INFO][QGenFullChecker] Exporting the MATLAB base workspace [INFO][QGenFullChecker] Creating the decoration file for model: speedometer.mdl [INFO][QGenFullChecker] Calculating Execution orders... [INFO][QGenFullChecker] Generating Decoration File... [INFO][QGenFullChecker] Determining model dependencies... [WARNING][QGenFullChecker] The model speedometer contains callback functions. Please make sure that none of them changes the model behavior or structure. [INFO][QGenFullChecker] Here is the list of blocks containing callback functions: speedometer/Compare To Constant [INFO][QGenFullChecker] Invoking QGen to check for model compatibility [WARNING][QGenFullChecker] QGen encountered errors while handling the model: [INFO] Opening M-file: ./slprj/qgen_checker/workspace.m [INFO] Loading library './lib.mdl' for block 'lib/Mod16_Difference' [ERROR] Could not import block speedometer/Polynomial: Block type 'Polyval' not supported [INFO] Preprocessing model speedometer [MISRA VIOLATION] MISRA AC SLSF 017: A model must not contain any unconnected block Block : Delay block speedometer/Delay Unconnected OutDataPort: OutDataPort1 [ERROR] Delay block speedometer/Delay: expected value 'Dialog' for parameter 'DelayLengthSource', found 'Input port' [ERROR] Delay block speedometer/Delay: structure check failed. [INFO] Finished preprocessing model speedometer [INFO] Sequencing model speedometer [INFO] Finished sequencing model speedometer [INFO][QGenFullChecker] Finished the full compatibility check

Fig. 4: Compatibility Report with Full Check level

4.4 Invocation from the MATLAB Command Line

The Compatibility Checker can be invoked from the MATLAB command line as follows:

```
qgen_compatibility_check(mdl_path [, subsys [, level [, report_format, [, verbosity]]]])
```

The first argument `mdl_path` is the path to the Simulink model file.

Optional arguments may follow in the following order:

- `subsys`: path to a subsystem in the model (Default: `''`)

  Restricts the check to the specified subsystem.

- `level`: one of the strings `'quick'`, `'datatypes'` or `'full'` (Default: `'quick'`)

  Select the level of checks to perform.

- `report_format`: one of the strings `'html'` or `'text'` (Default: `'html'`)

- `verbosity`: an integer between `0` and `2` (Default: `0`)

  Configure the level of verbosity in the messages displayed on the console. A higher value yields more detailed messages.

Example

For example, to perform the data types and callbacks check on the model `speedometer.mdl` and get a text report, use the following command:

```
>> qgen_compatibility_check('speedometer.mdl', '', 'datatypes', 'text')
[INFO][QGen] Initializing QGen Compatibility Checker for level DATATYPES
[INFO][QGen] (1/1) Loading speedometer
[INFO][QGen] (1/1) Checking speedometer
[INFO][QGen] (1/1) Running 'Update Diagram' of speedometer
[INFO][QGen] (1/1) ############
[INFO][QGen] (1/1) # Results #
[INFO][QGen] (1/1) ############
[INFO][QGen] (1/1) Found 2 constraint violations
[INFO][QGen] (1/1) 1/2 Check failed: The following blocks have types that are␣
↪unsupported by QGen
[INFO][QGen] (1/1)     1 offending elements:
[INFO][QGen] (1/1)       speedometer/Polynomial of type Polyval
[INFO][QGen] (1/1)     Workaround: Use (an)other supported block type(s) that provides␣
↪the same behavior
[INFO][QGen] (1/1) 2/2 Check failed:
[INFO][QGen] (1/1)     1 offending elements:
[INFO][QGen] (1/1)       speedometer/Compare To Constant
[INFO][QGen] (1/1)     Workaround: Make sure that the listed callback functions do not␣
↪modify the structure of the model
[INFO][QGen] (1/1)     Details:
[INFO][QGen] (1/1)      speedometer/Compare To Constant callback Initialization
[INFO][QGen] (1/1)      speedometer/Compare To Constant callback ZeroCross
[INFO][QGen] (1/1) Finished checking compatibility
```

4.5 Invocation from Model Advisor

If you have a valid *Simulink Check* license[1] , you can also use the QGen Compatibility Checker from Model Advisor.

To do so, open Model Advisor from the Simulink *Analysis* menu.

As shown in the screenshot, QGen's compatibility checks are integrated in Model Advisor as individual checks

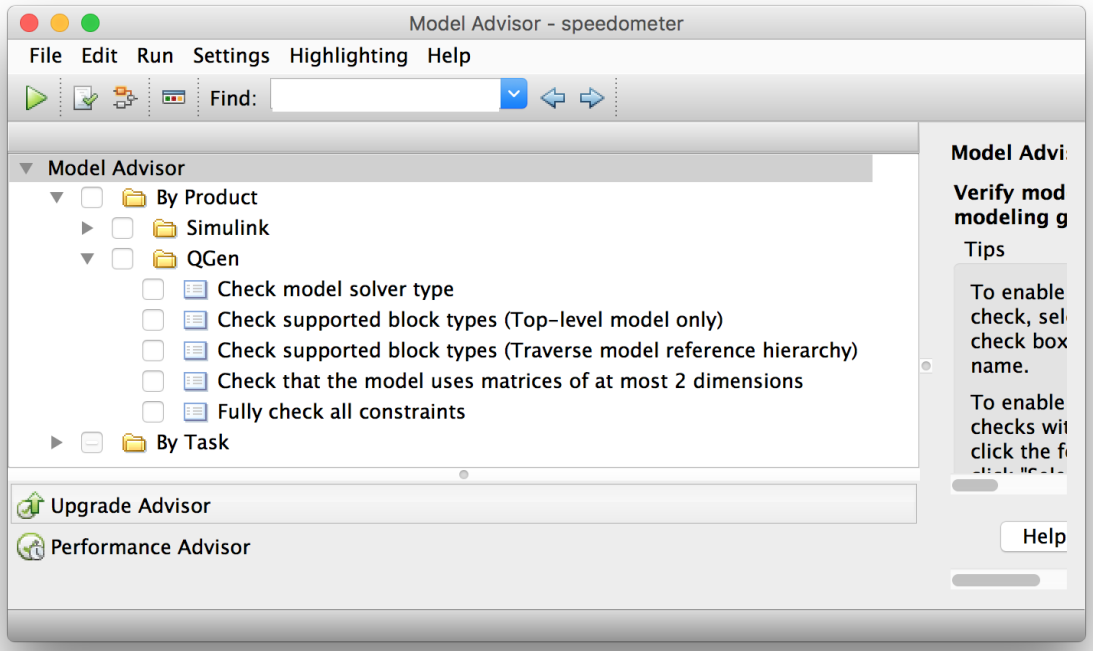


Fig. 5: QGen Compatibility Checker integration with Model Advisor

Select the checks you wish to perform, and click *Run Selected Checks*. The results of the checks are reported within the Model Advisor window as customary with standard Model Advisor checks.

[1] Contact MathWorks to purchase a *Simulink Check* license (named *Simulink Verification and Validation* before R2017b)

CHAPTER

FIVE

QGEN MODEL DEBUGGER

5.1 Requirements and installation

QGen Debugger is an add-on for QGen that relies on GNAT Studio to provide model debugging functionalities. Thus, before installing QGen Debugger, first proceed with the installation of GNAT Studio and QGen. See *Installing QGen* for more information about the installation of QGen.

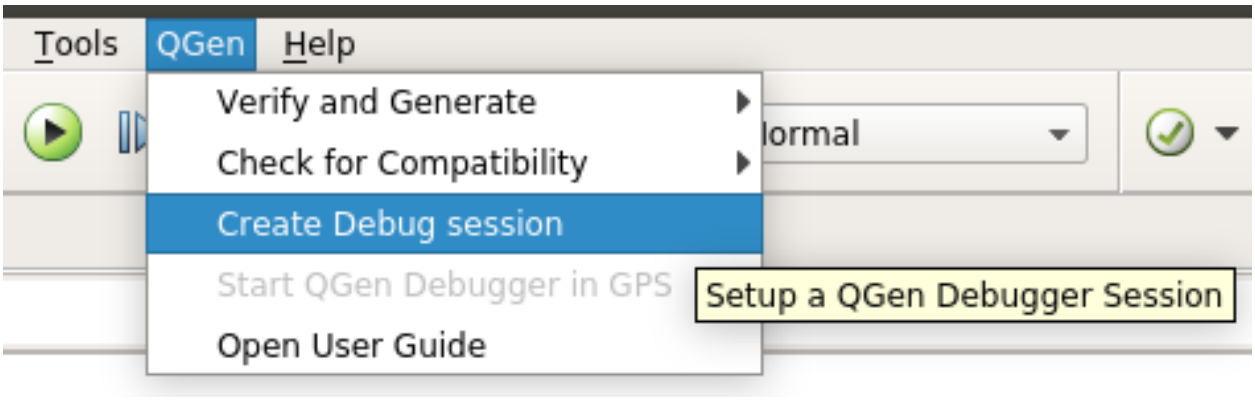
The QGen Debugger package contains multiple tools including a compiler and a debugger that are not intended to be used independently from the QGen Debugger and are not supported outside of that usage.

To proceed with the installation of QGen Debugger, follow the same instructions as for QGen in *Installing QGen*. Choose the QGen installation folder `$QGEN_INSTALL` as the installation directory.

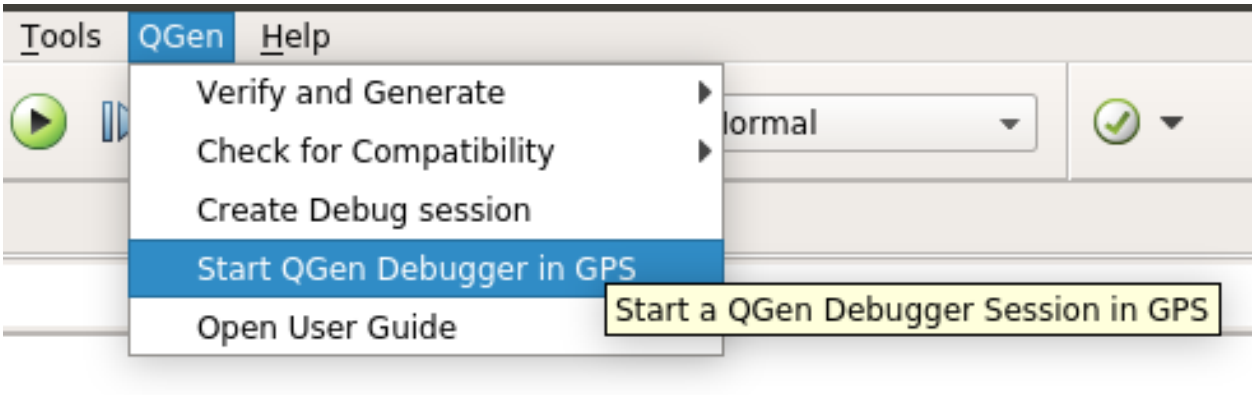
Then, to use QGen Debugger from your MATLAB environment, follow the setup process in *Setting up QGen in MATLAB*.

5.2 Create a Debugging session from Simulink

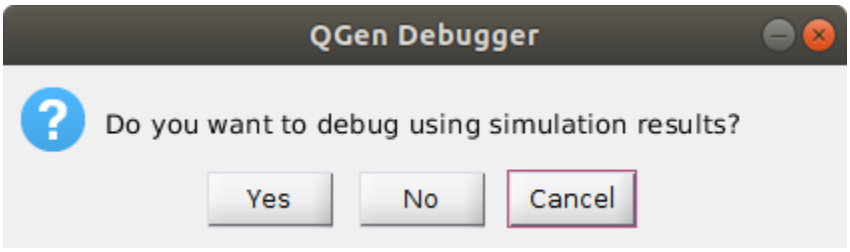
The easiest way to setup a debugging session is to use the project generation interface from Simulink. Open the model that you want to debug and click on *QGen > Create Debug session*.



This will generate a simulation model sending values to the original model, referenced using a Model block. You can alter the values inside the *inputs* subsystem using any Simulink block or block combination. By default, a Signal Builder block is created within the subsystem that lets the user customize the inputs. Once you are satisfied click on *QGen > Start QGen Debugger in {GPS|GNAT Studio}*.



QGen will ask whether you want to use the simulation results or not.



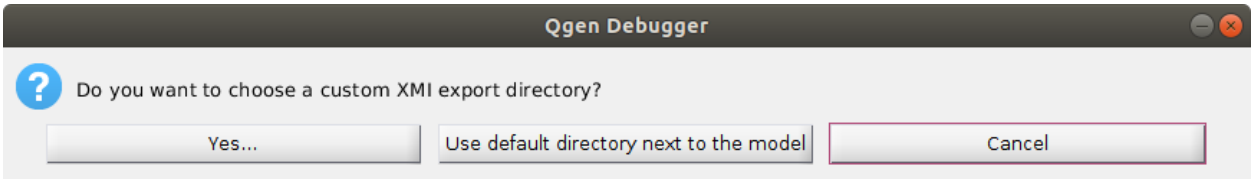
- If you select *Yes*, simulation inputs and outputs data will be saved to a file located at:

  `<WORKING_DIR>/sim_<MODEL_NAME>/<MODEL_NAME>.output.sl.txt`

  This input data will then be used as input when debugging the model with QGen Debugger.

- If you select *No*, the simulation data will not be exported and used by QGen.

The next popup will ask the user to select the output directory to export the model and the base workspace under the XMI format. Refer to *Exporting the model information* for more information.



If you choose to use the default directory next to the model, QGen will create the following directory to store the XMIs:

`<WORKING_DIR>/<MODEL_NAME>_qgen_dbg_xmi`

When the export step is done, QGen will start GNAT Studio (or the older GPS if specified in the PATH) and open a new project running your model with the chosen values. After starting, GNAT Studio will ask you what language you want to generate code for.

A project will be automatically created and loaded, feel free to edit it with additional options if needed, see *Setting up manually a Simulink project with GNAT Studio*.

Once the project has been created once, you can launch GNAT Studio and load it manually without running MATLAB beforehand.

You can now jump to the *Generating code and starting a debugging session* section.

5.2.1 Recommended workflow

- The first time to debug the model:

  Follow the steps above to create the simulation model, log the simulation results, export the model, generate a debugging project and open it in GNAT Studio before finally debugging the model with QGen Debugger.

- When the model has not changed:

  If the model and its dependencies and the workspace have not changed, it is not necessary to repeat the above procedure. Instead, start GNAT Studio (it is provided with QGen Debugger at `$QGEN_INSTALL/libexec/qgen/bin/gnatstudio[.exe]`) and open the debugging project `<MODEL_NAME>.gpr`. To start debugging, there is no need to re-generate the code with QGen, simply start the debugger with *Debug -> Initialize -> main* in the top menu.

- When the model has changed:

  The above standard procedure (*Create a Debugging session from Simulink*) applies. However, it may be a bit slow and tedious to do repeatedly.

  An alternative, faster approach is to re-export the XMI manually in MATLAB using:

```
qgen_export_xmi ('<MODEL_NAME>', '-o', '<OUTPUT_DIR>')
```

  where `<OUTPUT_DIR>` is the directory containing the exported XMI for the model. Then, start GNAT Studio independently, open the debugging project `<MODEL_NAME>.gpr`, re-generate the code from the new XMI and start debugging as described in *Generating code and starting a debugging session*.

Note: This approach does not update the simulation results. If you are interested in updating the simulation results after changing the model, use the standard procedure in *Create a Debugging session from Simulink*.

5.3 Setting up manually a Simulink project with GNAT Studio

To be able to use the Model Debugger you need a minimal configuration in a .gpr file in your project directory. You must at least specify the following properties (we target the C language here but you can replace it with Ada without any additional difference):

```
project Project_Name is

  -- Simulink must be added to the list of Languages
  for Languages use ("C", "QGen");

  package Compiler is
    -- The target executable has to be compiled with debug infos
    -- -fpreserve-control-flow is recommended to make sure breakpoints
    -- are properly set before assignments execution
    for Switches ("C") use ("-g", "-fpreserve-control-flow");
  end Compiler;

  -- A main file calling the generated code top level init and comp routines
  for Main use ("main.c");
  for Source_Dirs use ("src", "generated");
  for Object_Dir use "obj";
```

(continues on next page)

(continued from previous page)

```
  -- All the qgenc options are specified within this package
  package QGen is

    -- Generated files are generated inside the following directory
    for Output_Dir use "generated";

    -- The Target property specifies model that have to get their code
    -- generated for a given target
    for Target ("main.c") use ("my_model.xmi");

    -- We target the c language, the --debug option is not mandatory
    -- but disables optimization to provide better traceability between
    -- the model and the generated code
    for Switches ("my_model.xmi") use ("--debug", "-l", "c",
                  "--pre-process-xmi",
                  "--gen-entrypoint");
  end QGen;

end Project_Name;
```

The project directory corresponding to the previous gpr file will be:

```
./
./project_name.gpr
./generated/
./obj/
./model.slx
./src/
./src/main.c
```

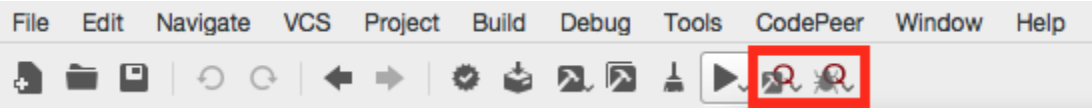
A project file generated by creating a debug session from MATLAB will contain a *Debug_Args* attribute in the *QGen* package, it is used to store the mandatory arguments for the framework to start correctly.

When launching {GNAT Studio|GPS} you can then load the project with Project > Open and browse to the project_name.gpr file.

You can find a working project file example in the QGen examples called `speedometer/speedometer.gpr`. You can open the project with GNAT Studio and start a debugging session by following the sections below.

5.4 Generating code and starting a debugging session

Once the project is successfully loaded inside GNAT Studio you should see the following icons:

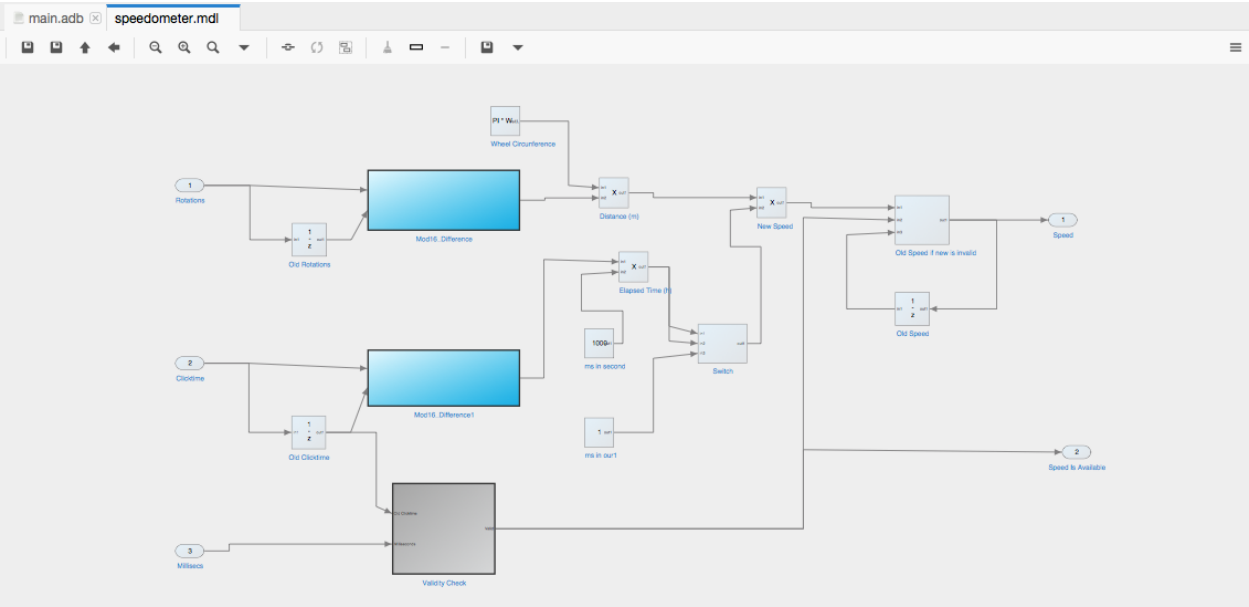


The left one will generate code using qgenc and the options specified inside the project file and then compile the executable using the generated code and any additional source file provided by the project file.

The right icon will compile like the left one but will also start a debugging session.

If you do not wish to generate code, but still need to build your executable and/or debug the model you can use the GNAT Sutdio menus *Build > Project > Build all* and *Debug > Initialize > [project_name] > main.[c|adb]*.

If you open the model by double cliking on it gps will load it and open it in a new view, displaying all the blocks and signals (cf. image below)



5.4.1 Editing QGen code generation options

To edit the options that QGen uses for code generation, there are two options:

Graphically

In the Project View, right-click on the XMI file that corresponds to the top-level model of the project (e.g. `speedometer.xmi`), then click on *Edit switches for speedometer.xmi*.

A configuration dialog will open. You may select options graphically or edit the command-line at the bottom. Clicking on *Save* will apply those switches to the XMI file and automatically update the project.

Editing the project source file

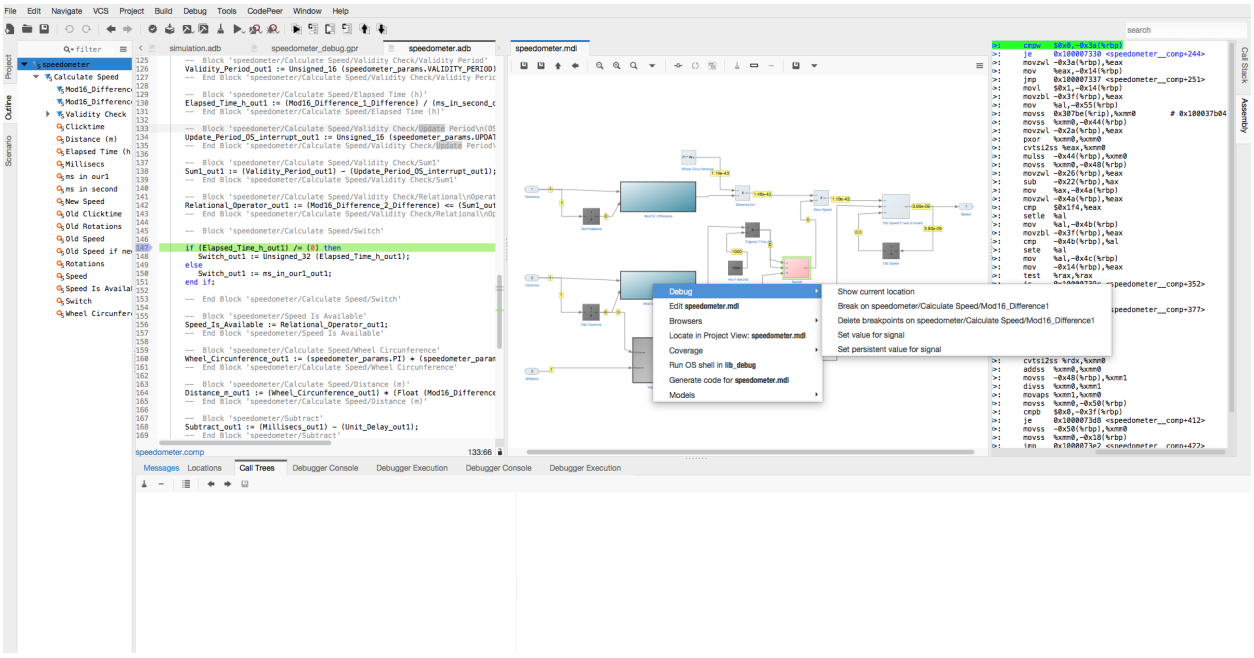
Alternatively, you may open the project source file and edit it. Right-click on the project in the Project View, *Project -> Edit source file*, then edit the following line to use the desired code generation options:

```
for Switches ("speedometer.xmi") use ("--debug", "-l", "c", "--pre-process-xmi", "--gen-
↪entrypoint");
```

Do not forget to reload the project after saving the file.

5.5 QGen Debugger features

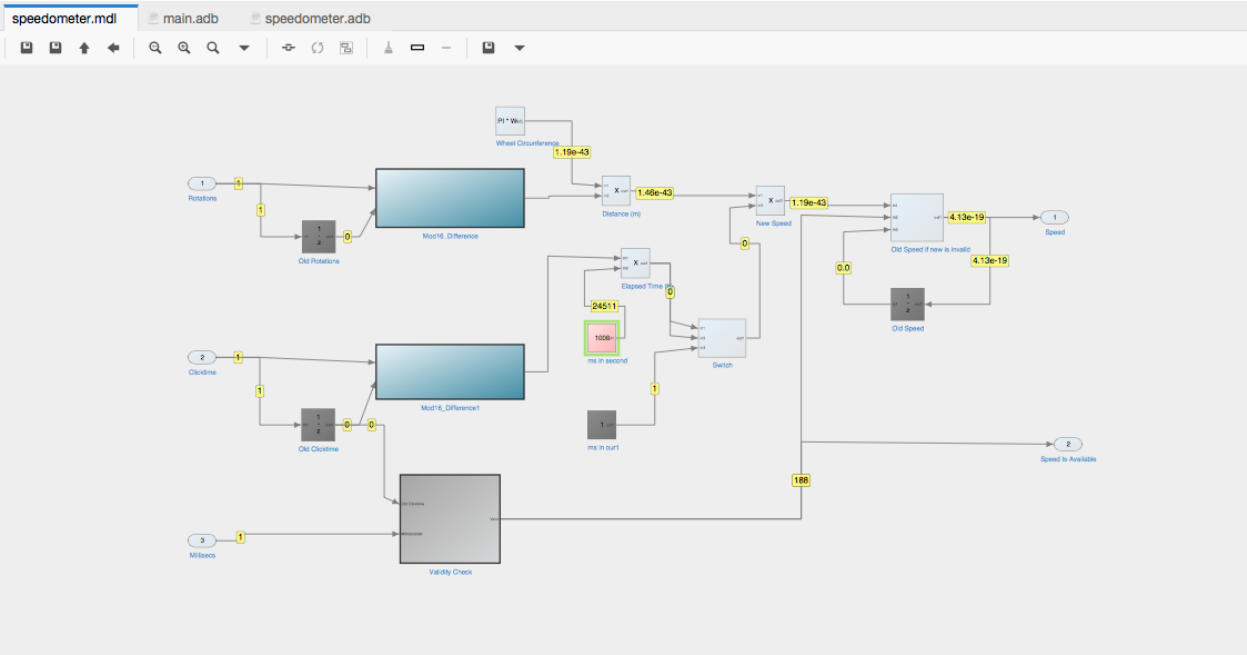
Once the debugger has started you can right click in the model view to display a contextual menu giving you access to the model debugging features.



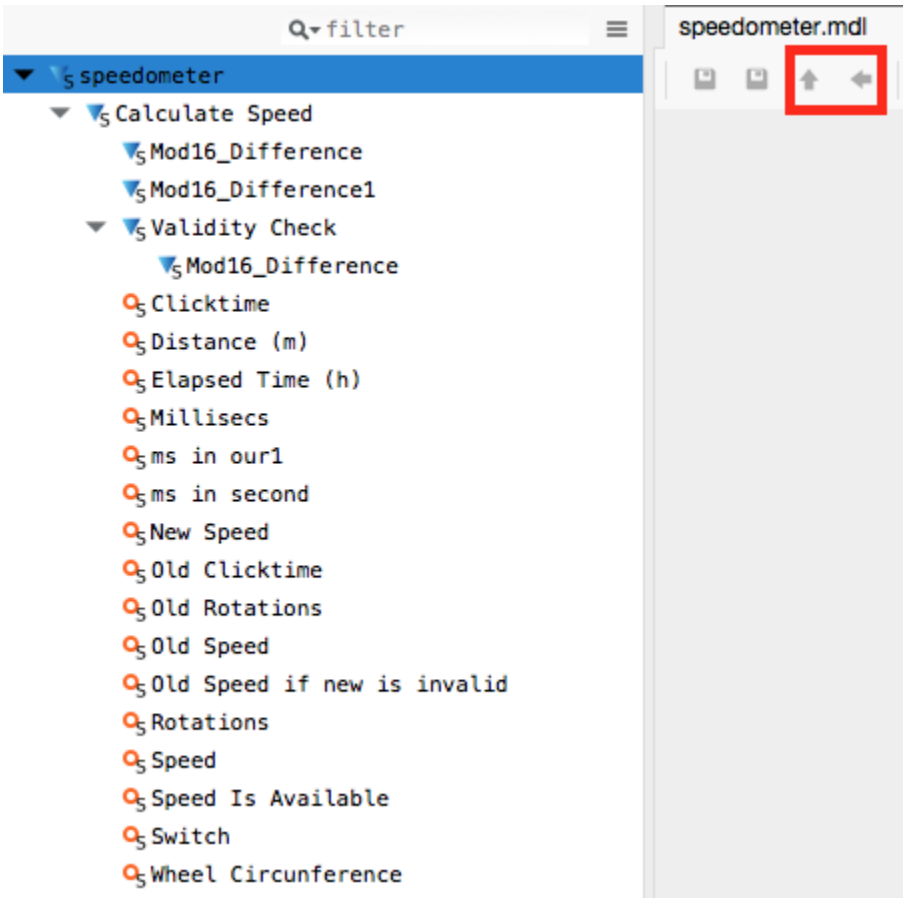
In the contextual menu hover on the `Debug` submenu to choose between the following options during debugging session

1. Show the current execution pointer in the source code
2. Set/Delete a breakpoint on the block (the corresponding breakpoints will be set or deleted inside the generated source code at the lines corresponding to the chosen block).
3. Change the value of a signal to alter the corresponding variable value in the code by clicking on `Set Value for signal`. This is really valuable to force a model to reach a certain stage and test it with a variety of values. You can force a signal to stay at a given value by checking the `Peristent` box and enter the desired value in the dialog box. This will preserve the value set over the program iterations to force specific cases to happen, which is a powerful way to test behaviours that are difficult to construct from the top level inputs.
4. In the `Models` submenu the `Show source code` action will show the block corresponding source code in the generated source file.

During the execution the model view is also updated to show the current block under execution with a green outline, and the value for all the signals in the current subsystem. Some signal values will not be up to date at some point during the execution but you can refer to the source code to find out where this is the case.



You can navigate between subsystems by double clicking on a subsystem or using the Uparrow and LeftArrow buttons above the diagram to navigate to the parent or previous subsystem respectively. The model hierarchy is also represented in the outline panel when the diagram is under focus, and you can click on a subsystem to visit it. All current blocks from the current subsystem are also listed in the outline panel, clicking on a block will highlight item.



All signals values are updated when the displayed subsystem changes.

5.5.1 QGen Debugger constraints

Stateflow charts

QGen Debugger only displays Stateflow Chart Blocks. They cannot be opened to display their internal events, states or transitions, and the debugging support is limited to setting breakpoints on the entry point of a chart.

5.6 Advanced usage

5.6.1 Use a separate compiler/debugger/{GNAT Studio|GPS} version

You might be interested in using a different version of the tools packaged with the QGen Debugger. This is possible but please be aware that some features of the debugger might not work with older versions of the tools.

To specify which tools to use when launching {GNAT Studio|GPS} and the QGen debugger from MATLAB you must edit the file *etc/qgen_tools.env* in the QGen installation path. You can either add each necessary environment variable manually or on Unix*, simply run *env >> etc/qgen_tools.env* from your terminal.

To change the compiler/debugger within {GNAT Studio|GPS} go to *Build* > *Settings* > *Toolchains* and edit the *Compiler path* and *Tools path* as necessary.

CHAPTER

SIX

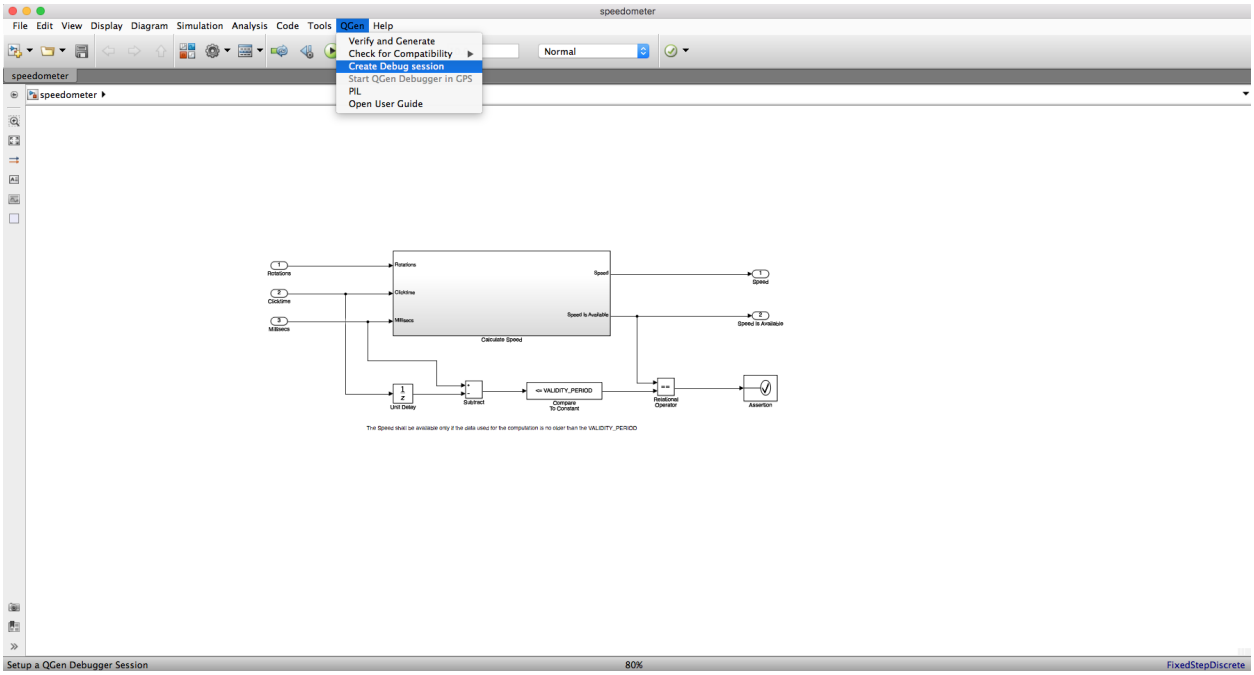
QGEN MODEL DEBUGGER: STEP BY STEP TUTORIAL

6.1 General comments

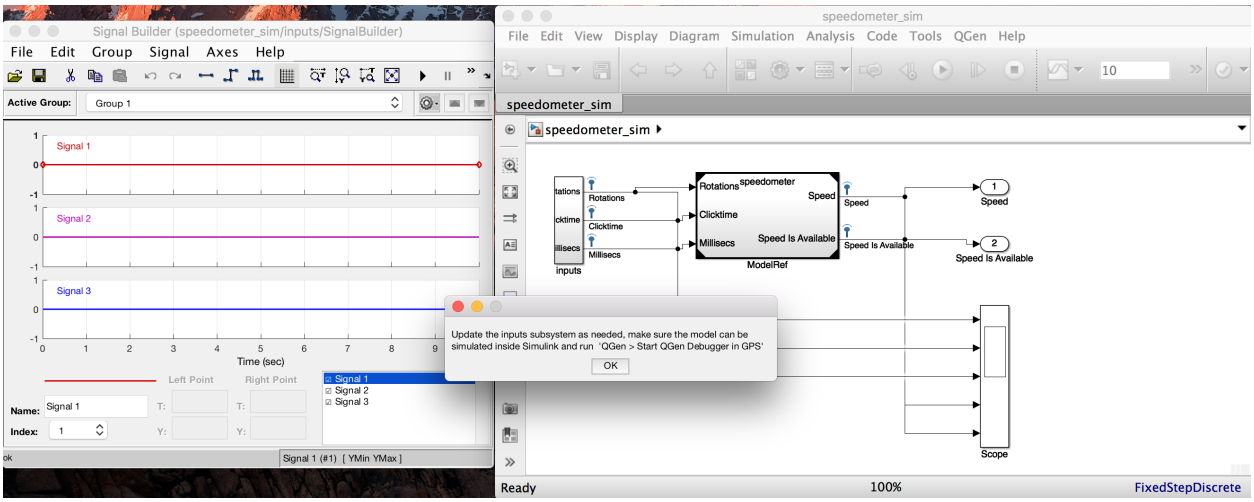
For installation and general documentation please refer to *QGen Model Debugger*. This tutorial assumes QGen and QGen debugger have been setup succesfully, see *Installing QGen* for more details.

6.2 Create a Debugging project from MATLAB

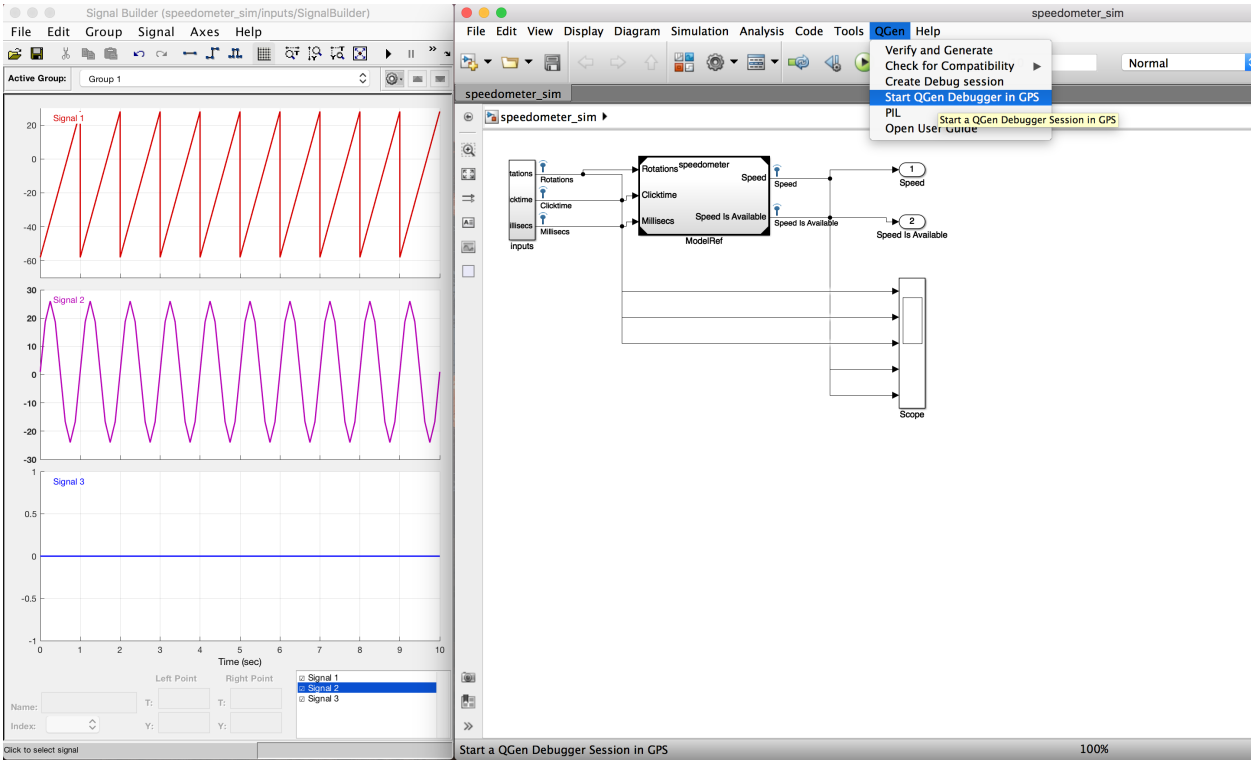
Open the model that you want to debug in MATLAB, in this example we will be using the model available *QGEN_INSTALL/share/qgen/examples/speedometer/speedometer.mdl*, and click on *QGen > Create Debug session*. This will create a new harness model in the same directory called *<model_name>_sim.slx*.



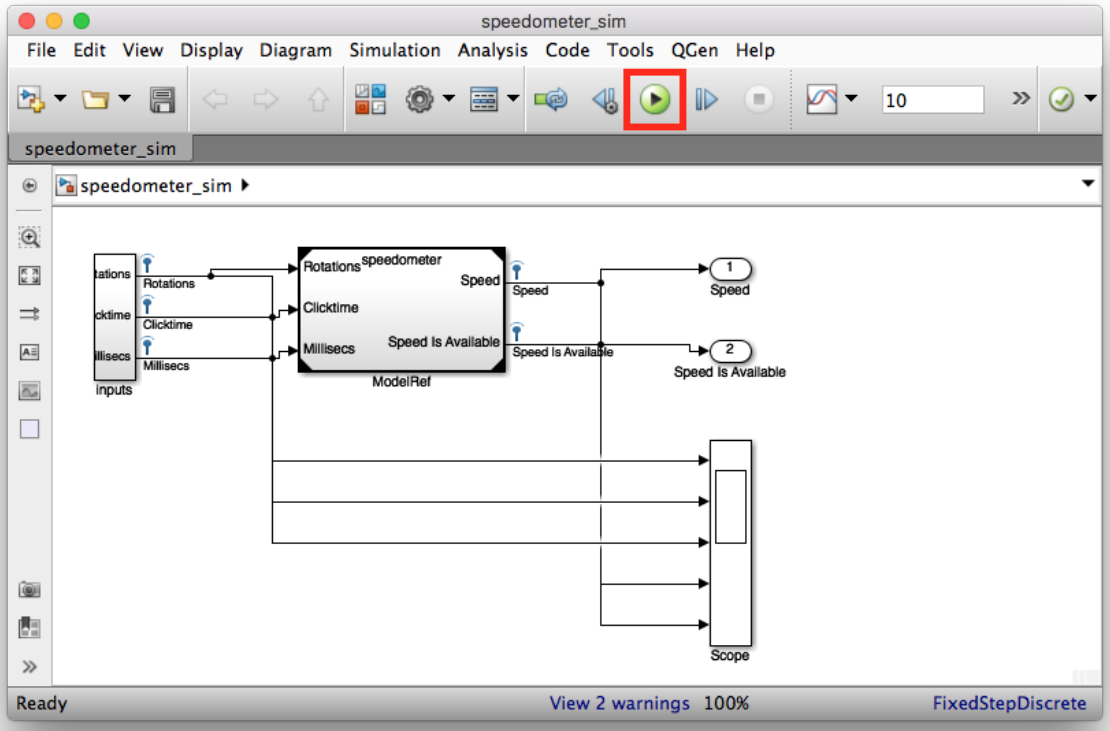
This model references the original model and logs the signals connected to its input and outputs ports. Additionally, it creates an *inputs* subsystem that you can use to generate an appropriate test case. Click *OK* on the dialog window that appeared, and proceed by updating the signals within the *Signal Builder* window. Please refer to the signal builder documentation for more detail.



Feel free to replace the signal builder block by any combination of blocks or model references if desired.

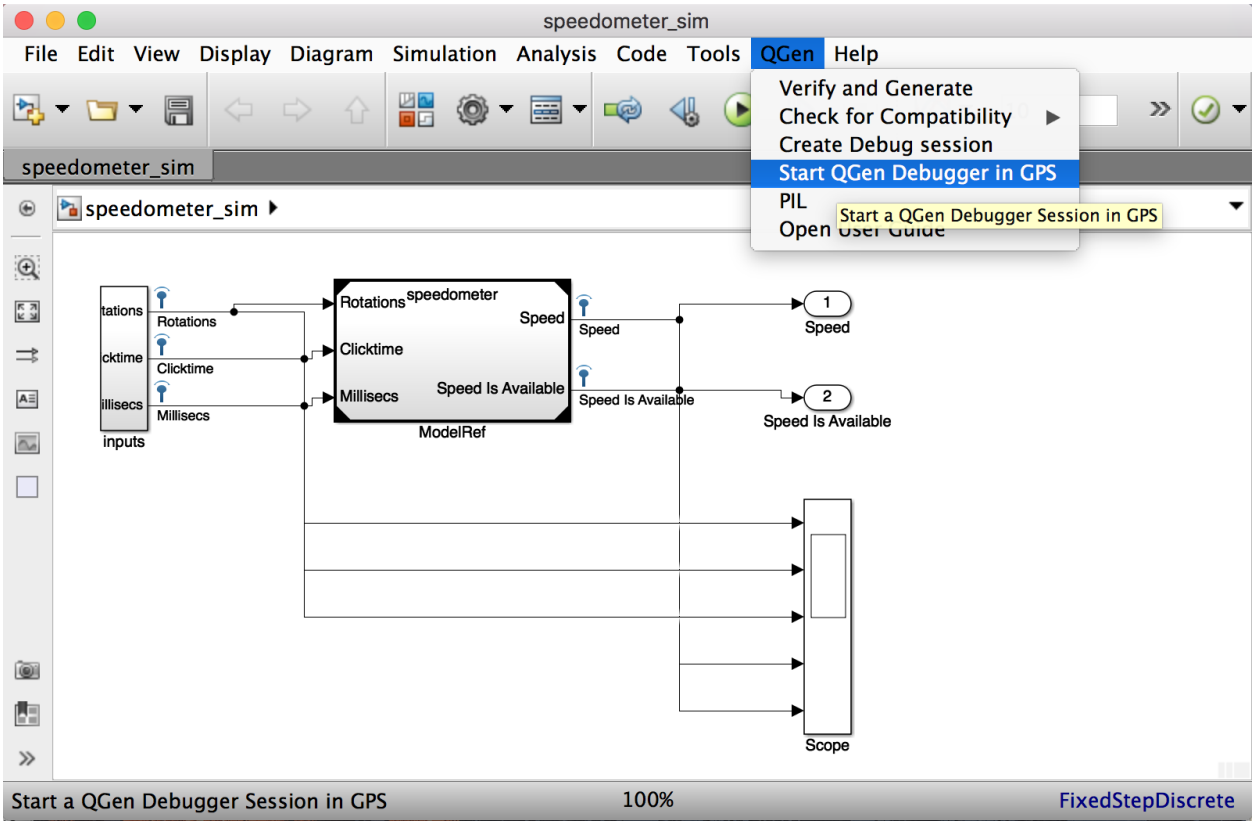


Once the test case is ready, run the simulation on your model by clicking on the icon highlighted in red below.

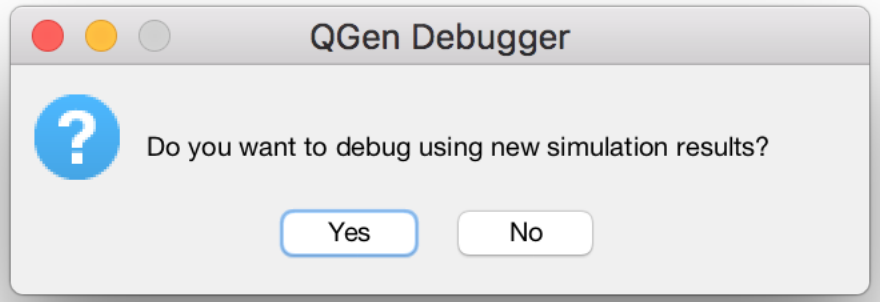


6.3 Start Debugging the model in GNAT Studio

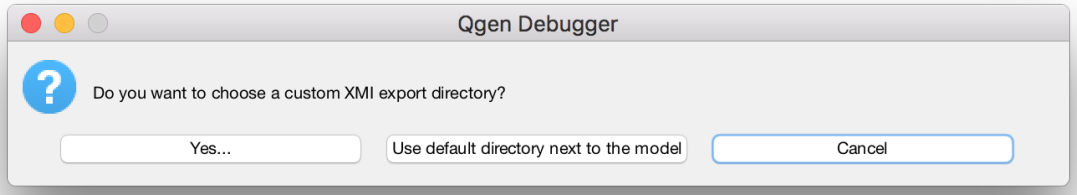
Once you can simulate the model successfully, click on *QGen* > *Start QGen Debugger in GNAT Studio*.



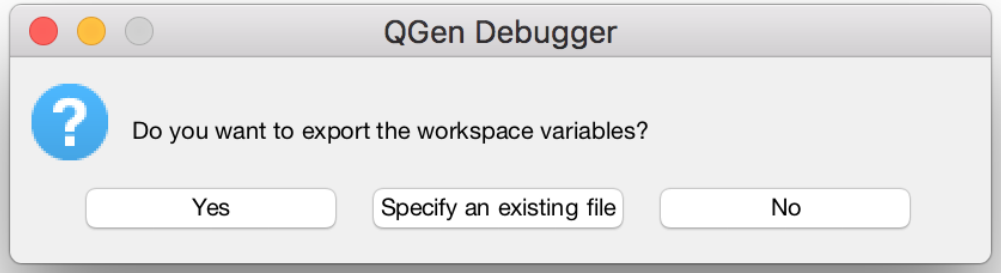
Two dialog windows will appear to define the debugging environment. The first will ask you whether you want to use the test case created previously, click on *Yes* if you want to use the Simulink inputs to call your generated code. If you click on *No*, a simple main file will be generated and you will be able to write the calling code manually. This is the recommended choice for more complex integrations with existing handwritten code.



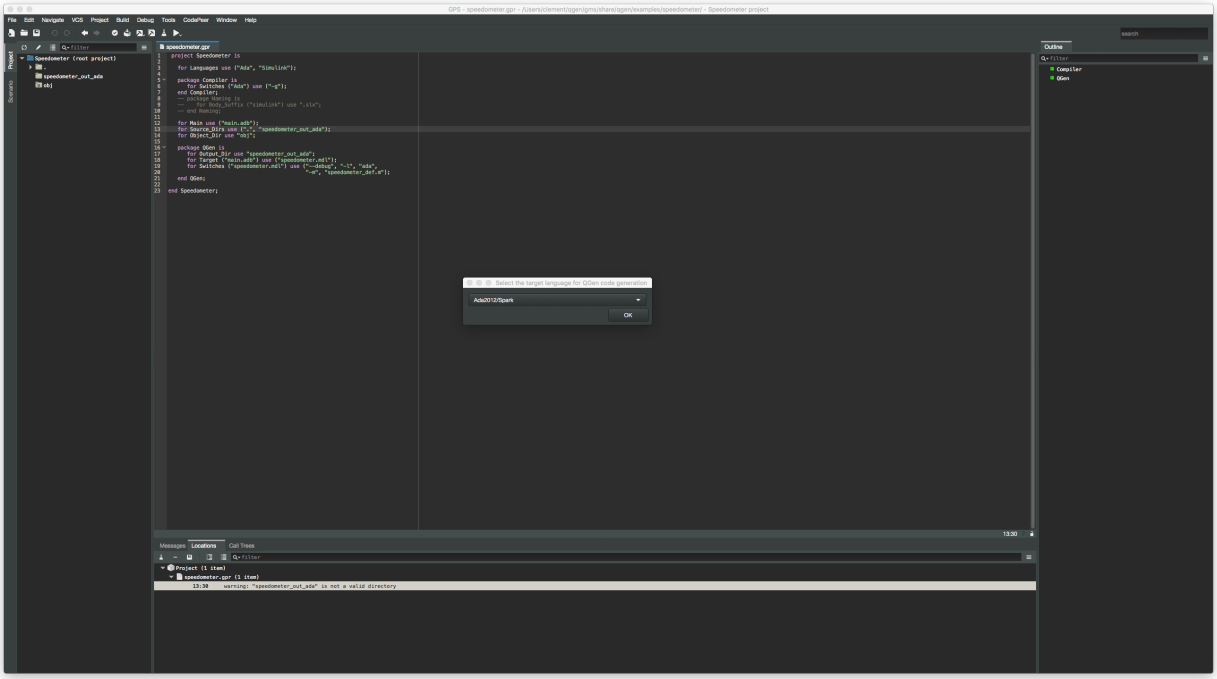
You can choose a directory to store the exported XMI files, this allows you to target an existing export directory to speed up the process.



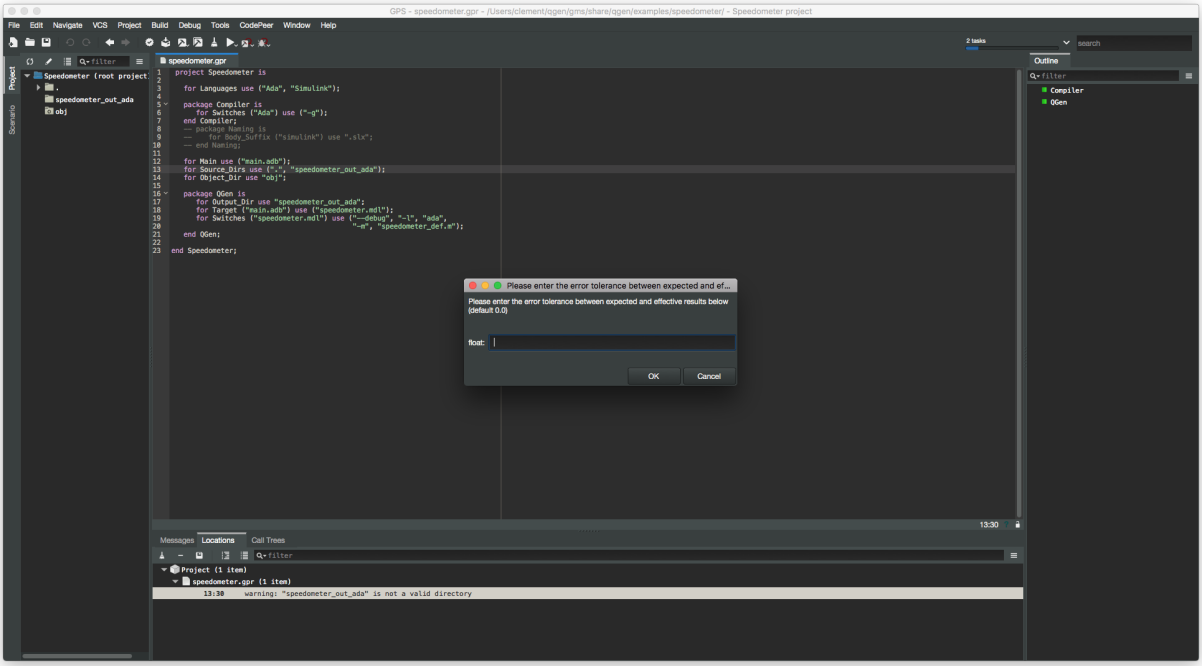
If your model is using workspace variables click on *Yes* in the next dialog.



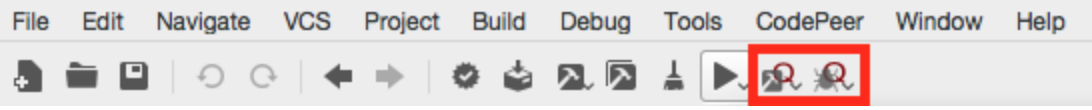
GNAT Studio will then open and ask you to choose a target language for code generation, pick *Ada* or *C* and click on *OK*.

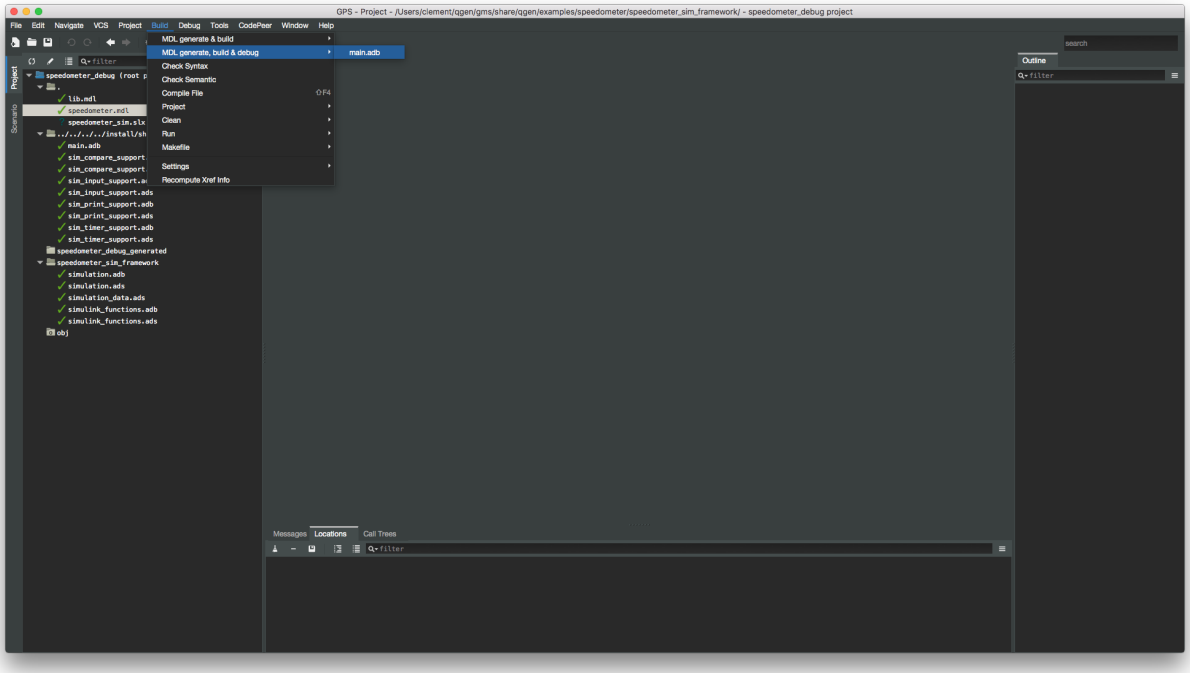


If you selected *Yes* at the first step. The next pop-up will ask you about a tolerance value that is used to consider the simulation correct or not. All output values coming from the generated code will have to be inside the range [simulated-value - tolerance, simulated-value + tolerance] to be considered correct. Choose a value suiting you needs or leave it empty to use 0.0 as default.



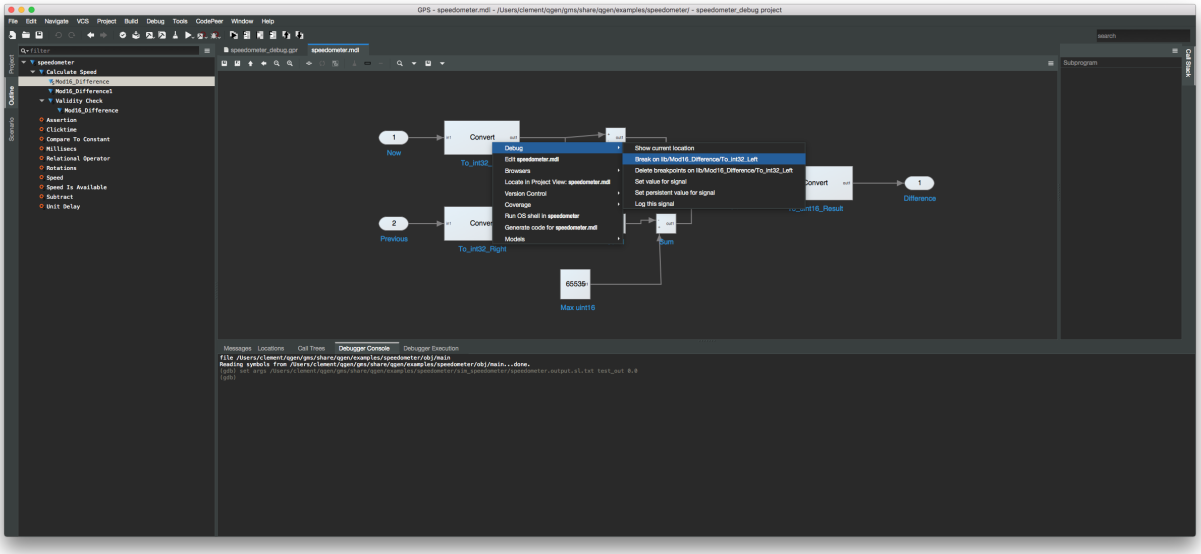
You can then generate code, build the executable and start the debugger by clicking on the right icon shown in the image below or by clicking on *Build* > *MDL generate, build & debug* > *main.adb*



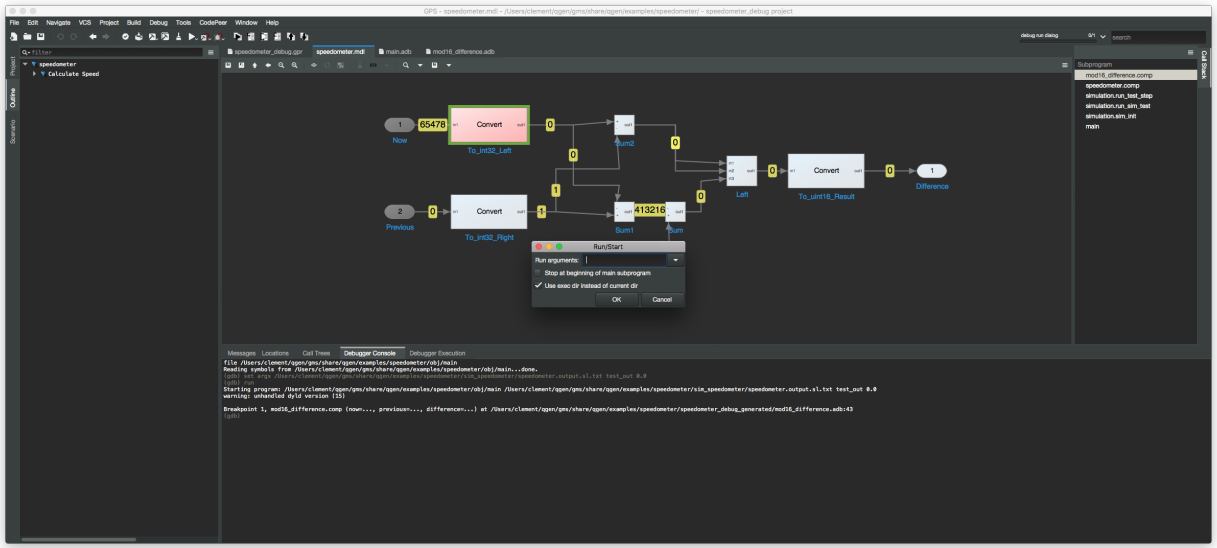


After the generation and build process complete a *Debugger Console* Panel will appear. Double click on your original model in the project view to display it in a new GNAT Studio view.

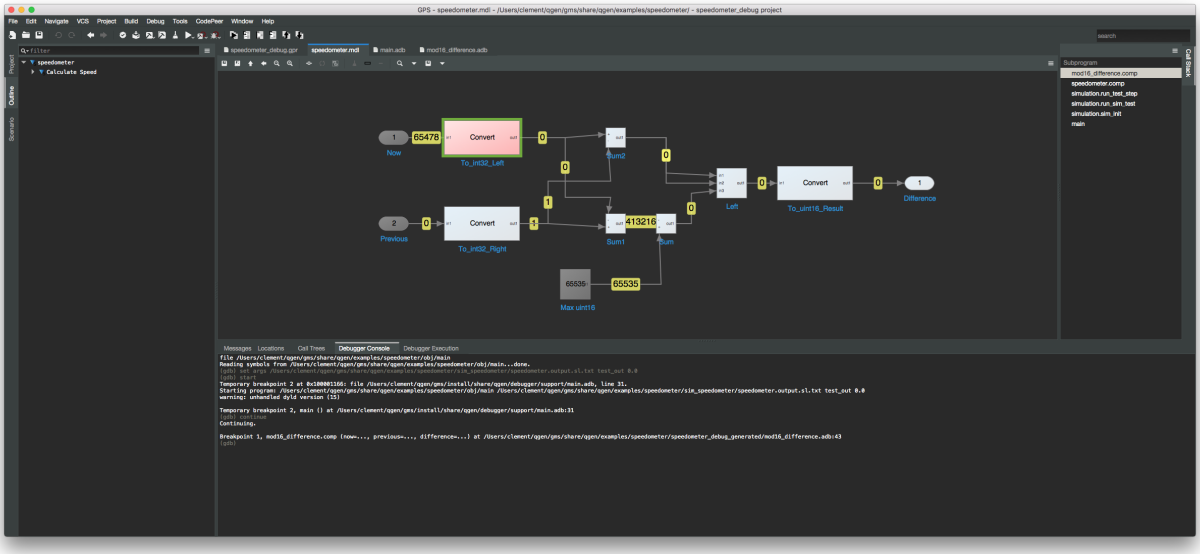
Once the model is open browse to the desired subsystem by double-clicking on a subsystem or by using the outline view on the left. Set a breakpoint on a block by right clicking on it and clicking on *Debug* > *Break on <block_name>*.



Click on *Debug* > *Run...*, uncheck *Stop at beginning of main subprogram* and click on *OK* to start debugging.



Execution will reach the breakpoint on the block defined earlier and you can examine the values and execute step by step. Please refer to *QGen Model Debugger* for all the functionalities available to you at that stage.



CHAPTER

SEVEN

INTEGRATING EXTERNAL CODE

7.1 Background

QGen supports interfacing generated code with manually written or code generated from another model by following means:

- Using custom data types defined in external modules
- Importing variables and constants defined in external modules
- Calling C or Ada code using S-Function block

7.2 Using custom data types defined in external modules

Procedures for handling custom datatypes is described in *Custom Data Types*

7.3 Importing variables and constants defined in external modules

Variable or constant defined in an external module can be imported using Simulink Signal or Simulink Parameter objects. Signal and Parameter objects with StorageClass 'ImportedExtern' or StorageClass 'Custom' and CustomStorageClass 'ImportFromFile' are treated as defined in an external module. In the case of StorageClass 'Custom', there is the possibility to indicate a specification file containing the symbol declaration. If a specification file is not defined, QGen generates an import declaration.

7.3.1 Imported variable with no explicit declaration

When importing a variable with StorageClass 'ImportedExtern' or in case the StorageClass is 'Custom' and the HeaderFile parameter is not defined, QGen assumes that the variable was defined in an external module with C convention. Variable declaration is derived from the type specification in Matlab.

```
% SIGNAL IntVarDefault %
% External varible, no header specified
% Assumed to be defined in C by default
IntVarDefault = mpt.Signal;
IntVarDefault.DataType = 'int32';
IntVarDefault.Min = [];
IntVarDefault.Max = [];
IntVarDefault.Description = '';
```

(continues on next page)

(continued from previous page)

```
IntVarDefault.RTWInfo.StorageClass = 'Custom';
IntVarDefault.RTWInfo.Alias = '';
IntVarDefault.RTWInfo.Alignment = -1;
IntVarDefault.RTWInfo.CustomStorageClass = 'ImportFromFile';
IntVarDefault.RTWInfo.CustomAttributes.ConcurrentAccess = 0;
% SIGNAL IntVarDefault %
```

When generating Ada code from the Signal object defined above the corresponding variable is declared in qgen_base_workspace package and imported with explicit "pragma Import" directive:

```
package qgen_base_workspace is
   IntVarDefault : Integer_16;
   pragma Import (C, IntVarDefault, "IntVarDefault");

end qgen_base_workspace;
```

When generating C, the variable is declared with "extern" qualifier in qgen_base_workspace.h

```
#ifndef QGEN_BASE_WORKSPACE_H
#define QGEN_BASE_WORKSPACE_H
#include "qgen_types.h"

extern GAINT16 IntVarDefault;

#endif
```

7.3.2 Imported variable with explicit declaration

Adding a reference to specification file, where external variable is declared, allows qgen to import that specification directly and declaration in generated code is not required.

```
% SIGNAL IntVarC %
IntVarC = mpt.Signal;
IntVarC.DataType = 'int16';
IntVarC.Min = [];
IntVarC.Max = [];
IntVarC.Description = '';
IntVarC.RTWInfo.StorageClass = 'Custom';
IntVarC.RTWInfo.Alias = '';
IntVarC.RTWInfo.Alignment = -1;
IntVarC.RTWInfo.CustomStorageClass = 'ImportFromFile';
IntVarC.RTWInfo.CustomAttributes.MemorySection = 'Default';
IntVarC.RTWInfo.CustomAttributes.DataAccess = 'Direct';
IntVarC.RTWInfo.CustomAttributes.HeaderFile = 'ext_signal_c.h';
%IntVarC.RTWInfo.CustomAttributes.ConcurrentAccess = 0;
% SIGNAL IntVarC %

% SIGNAL IntVarAda %
IntVarAda = mpt.Signal;
IntVarAda.DataType = 'int16';
IntVarAda.Min = [];
```

(continues on next page)

(continued from previous page)

```
IntVarAda.Max = [];
IntVarAda.Description = '';
IntVarAda.RTWInfo.StorageClass = 'Custom';
IntVarAda.RTWInfo.Alias = '';
IntVarAda.RTWInfo.Alignment = -1;
IntVarAda.RTWInfo.CustomStorageClass = 'ImportFromFile';
IntVarAda.RTWInfo.CustomAttributes.MemorySection = 'Default';
IntVarAda.RTWInfo.CustomAttributes.DataAccess = 'Direct';
IntVarAda.RTWInfo.CustomAttributes.HeaderFile = 'ext_signal_ada.ads';
IntVarAda.RTWInfo.CustomAttributes.ConcurrentAccess = 0;
% SIGNAL IntVarAda %
```

When generating Ada code, *ext_signal_ada.ads`* is included using `with` statement, an import pragma is defined in package `ext_signal_c` for variable declared in *ext_signal_c.h*. Note, that the initialization statement is generated only for variable `IntVarAda2` defined in the model. All imported variables are expected to be initialized in their originating modules.

```
package ext_signal_c is
   IntVarC : Integer_16;
   pragma Import (C, IntVarC, "IntVarC");

end ext_signal_c;
```

```
with ext_signal_c; use ext_signal_c;
with ext_signal_ada; use ext_signal_ada;
with qgen_base_workspace; use qgen_base_workspace;

package body ext_signal_c_ada is

   procedure initStates (State : in out ext_signal_c_ada_State) is
   begin
      --  Block 'ext_signal_c_ada/IntVarAda2Mem'
      State.IntVarAda2 := Integer_16 (0.0);
      --  End Block 'ext_signal_c_ada/IntVarAda2Mem'
   end initStates;

   procedure comp
      (Out1 : out Integer_16;
       Out2 : out Integer_16;
       Out3 : out Integer_16;
       Out4 : out Integer_16;
       State : in out ext_signal_c_ada_State)
   is
   begin
      Out1 := qgen_base_workspace.IntVarDefault;

      Out2 := ext_signal_c.IntVarC;

      Out3 := ext_signal_ada.IntVarAda;

      Out4 := State.IntVarAda2;
   end comp;
```

(continues on next page)

(continued from previous page)

```
end ext_signal_c_ada;
```

In case of C as the output language the pattern is the same, except that now we generate "extern" declaration for originally declared in *ext_signal_ada.ads* and insert include statement for the one declared in *ext_signal_c.h*.

```
#ifndef EXT_SIGNAL_ADA_H
#define EXT_SIGNAL_ADA_H
#include "qgen_types.h"

extern GAINT16 IntVarAda;

#endif
```

```
#ifndef EXT_SIGNAL_C_ADA_H
#define EXT_SIGNAL_C_ADA_H
#include "ext_signal_c_ada_states.h"
#include "qgen_types.h"
#include "qgen_base_workspace.h"
#include "ext_signal_c.h"
#include "ext_signal_ada.h"

extern void ext_signal_c_ada_initStates
  (ext_signal_c_ada_State* const State);
extern void ext_signal_c_ada_comp
  (GAINT16* const Out1,
   GAINT16* const Out2,
   GAINT16* const Out3,
   GAINT16* const Out4,
   ext_signal_c_ada_State* const State);

#endif
```

```
#include "ext_signal_c_ada.h"

void ext_signal_c_ada_initStates
  (ext_signal_c_ada_State* const State)
{
   /* Block 'ext_signal_c_ada/IntVarAda2Mem' */
   State->IntVarAda2 = (GAINT16) 0.0;
   /* End Block 'ext_signal_c_ada/IntVarAda2Mem' */

}
void ext_signal_c_ada_comp
  (GAINT16* const Out1,
   GAINT16* const Out2,
   GAINT16* const Out3,
   GAINT16* const Out4,
   ext_signal_c_ada_State* const State)
{
   *Out1 = IntVarDefault;

   *Out2 = IntVarC;
```

(continues on next page)

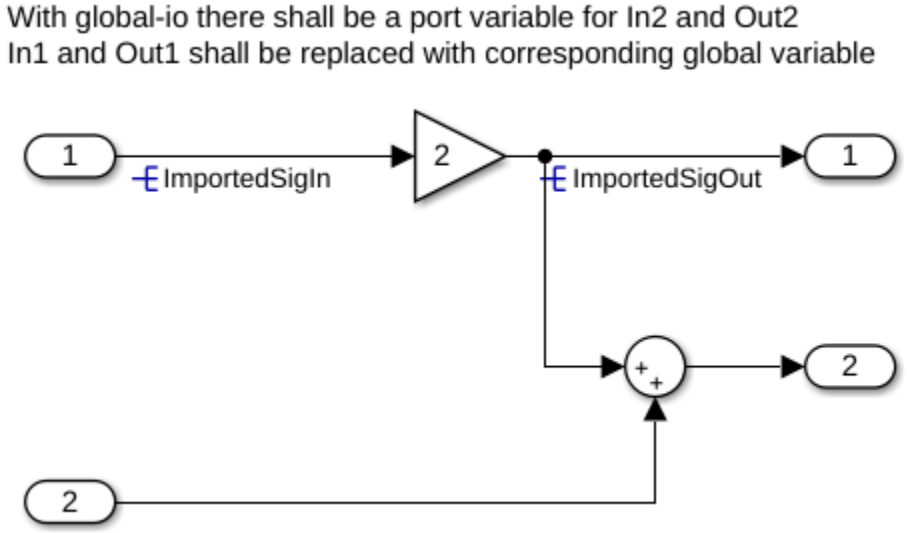
(continued from previous page)

```
    *Out3 = IntVarAda;

    *Out4 = State->IntVarAda2;
}
```

7.3.3 Imported variables in subystem interface

QGen supports several modes of interfacing with generated code (see full list of modes in section *Calling the generated code*). By default each top level port is converted to an argument of corresponding compute function. The `--global-io` switch changes this behavior and generates global variables instead.

If any of the input and output ports is connected to a signal with imported variable, the QGen-generated global variable is omitted and the imported one is used for IO.




```
% SIGNAL ImportedSigIn %
ImportedSigIn = Simulink.Signal;
ImportedSigIn.DataType = 'int16';
ImportedSigIn.Dimensions = [2 2];
ImportedSigIn.Complexity = 'real';
ImportedSigIn.Min = -100;
ImportedSigIn.Max = 100;
ImportedSigIn.SamplingMode = 'Sample based';
ImportedSigIn.SampleTime = 5;
ImportedSigIn.Description = 'Global input';
ImportedSigIn.RTWInfo.StorageClass = 'ImportedExtern';
% SIGNAL ImportedSigIn %

% SIGNAL ImportedSigOut %
ImportedSigOut = Simulink.Signal;
ImportedSigOut.DataType = 'int16';
ImportedSigOut.Dimensions = [2 2];
```

(continues on next page)

(continued from previous page)

```
ImportedSigOut.Complexity = 'real';
ImportedSigOut.Min = -100;
ImportedSigOut.Max = 100;
ImportedSigOut.SamplingMode = 'Sample based';
ImportedSigOut.SampleTime = 5;
ImportedSigOut.Description = 'Global output';
ImportedSigOut.RTWInfo.StorageClass = 'ImportedExtern';
% SIGNAL ImportedSigOut %
```

By default all ports have a corresponding function argument that, in case of an imported Signal object, is assigned to/from global variable

```
void simulink_signals_extern_io_comp
  (GAINT16 const In1[2][2],
   GAINT16 const In2[2][2],
   GAINT16 Out1[2][2],
   GAINT16 Out2[2][2])
{
   GAUINT8 i;
   GAUINT8 j;

   /* Block 'simulink_signals_extern_io/In1' */
   for (i = 0; i <= 1; i++) {
      for (j = 0; j <= 1; j++) {
         ImportedSigIn[i][j] = In1[i][j];
      }
   }
   /* End Block 'simulink_signals_extern_io/In1' */

   /* Block 'simulink_signals_extern_io/Gain' */
   for (i = 0; i <= 1; i++) {
      for (j = 0; j <= 1; j++) {
         ImportedSigOut[i][j] = 2 * ImportedSigIn[i][j];
      }
   }
   /* End Block 'simulink_signals_extern_io/Gain' */

   /* Block 'simulink_signals_extern_io/Out1' */
   for (i = 0; i <= 1; i++) {
      for (j = 0; j <= 1; j++) {
         Out1[i][j] = ImportedSigOut[i][j];
      }
   }
   /* End Block 'simulink_signals_extern_io/Out1' */

   /* Block 'simulink_signals_extern_io/Sum' */
   /* Block 'simulink_signals_extern_io/In2' */
   /* Block 'simulink_signals_extern_io/Out2' */
   for (i = 0; i <= 1; i++) {
      for (j = 0; j <= 1; j++) {
         Out2[i][j] = ImportedSigOut[i][j] + In2[i][j];
      }
```

(continues on next page)

(continued from previous page)

```
   }
   /* End Block 'simulink_signals_extern_io/Out2' */
   /* End Block 'simulink_signals_extern_io/In2' */
   /* End Block 'simulink_signals_extern_io/Sum' */
}
```

Running `qgenc` with `--global-io` switch removes the function arguments and generates global variables for In2 and Out2. External variables ImportedSignIn and ImportedSignOut are used for ports In1 and In2.

```
void simulink_signals_extern_io_comp (void) {
   GAUINT8 i;
   GAUINT8 j;

   /* Block 'simulink_signals_extern_io/Gain' */
   for (i = 0; i <= 1; i++) {
      for (j = 0; j <= 1; j++) {
         ImportedSigOut[i][j] = 2 * ImportedSigIn[i][j];
      }
   }
   /* End Block 'simulink_signals_extern_io/Gain' */

   /* Block 'simulink_signals_extern_io/Sum' */
   /* Block 'simulink_signals_extern_io/In2' */
   /* Block 'simulink_signals_extern_io/Out2' */
   for (i = 0; i <= 1; i++) {
      for (j = 0; j <= 1; j++) {
         simulink_signals_extern_io_comp_Out2[i][j] = ImportedSigOut[i][j] + simulink_
↪signals_extern_io_comp_In2[i][j];
      }
   }
   /* End Block 'simulink_signals_extern_io/Out2' */
   /* End Block 'simulink_signals_extern_io/In2' */
   /* End Block 'simulink_signals_extern_io/Sum' */
}
```

7.4 Calling C or Ada code using S-Function block

Launching legacy code in Simulink is achieved using S-Function blocks and mex files. Mex file (MATLAB EXecutable, http://www.mathworks.se/help/matlab/matlab_external/introducing-mex-files.html) is a compiled binary format, proprietary to Mathworks. The legacy code written in C, C++, Fortran or Ada can be compiled to this format and then linked to a model through S-Function blocks (called "Legacy Function" blocks).

For executing simulation no information on original sources is required. In case a model containing Legacy S-Function blocks is used for code generation in Real Time Workshop one needs to provide a special template file for each mex function containing the function prototype, reference to file containing function prototype etc. `qgenc` does not support this template format and relies only on the information contained in mdl and slx files.

The only type of supported S-Function is "Legacy function". There are two options to generate code from this block:

- Encode all information about the external library in S-Function parameters using the legacy_code tool and let `qgenc` generate the calls or
- Generate a wrapper function using `qgenc` and write the actual call to the legacy function manually in this wrapper.

7.5 Option A: specify library information in Simulink

This option assumes that the external library with legacy code is fully specified in S-Function parameters. At minimum the information we need is the name of the header file of the external library and a function signature specifying the mapping to block ports and data types of each argument.

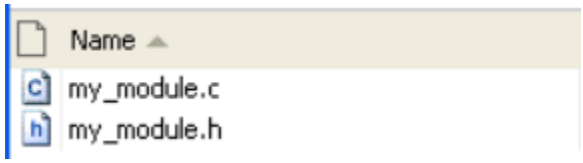
The workflow consist of the steps as follows:

- create S-Function block using the legacy code tool (see instructions below)
- generate code using QGen. The generated code will contain call to the function in external library
- while compiling the code provide the legacy code module in include and lib paths

7.5.1 Creating S-Function blocks in Simulink

There are several methods for creating Legacy Function blocks in Simulink. The only method, that stores all information required for generation is that using the Legacy Code Tool (http://www.mathworks.se/help/simulink/slref/legacy_code.html or http://www.mathworks.se/help/rtw/block-authoring-with-legacy-code-tool.html).

Let us consider creating a block for function "double myFun (in1 double)" with specification in file my_module.h:




```
#ifndef MYMODULE_H
#define MYMODULE_H

double myFun (double in1);

#endif
```

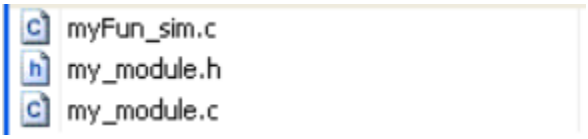
The first step is to create a datastructure introducing this function to the legacy code tool:

```
% define name of the function in Simulink (and name of the mex file)
% this name does not need to be the same as the referenced legacy function
def.SFunctionName = ('myFun_sim')

% define source file (body)
def.SourceFiles = {'my_module.c'}

% define header file (spec)
% NB! This file is not required for successful generation of mex files
% however, it is mandatory for linking the generated code to correct module
def.HeaderFiles = {'my_module.h'}

% define function prototype (see syntax of the prototype specification in the
% next section)
def.OutputFcnSpec = 'double y1 = myFun (double u1)'
```

To generate a block Simulink is able to execute we run the legacy code tool

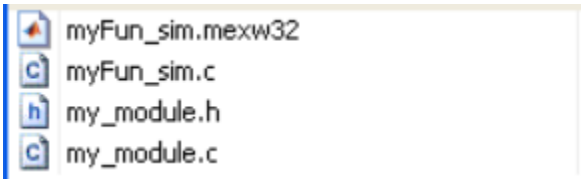
```
% generate wrapper for the legacy code
% this will create file 'myFun_sim.c' which is a wrapper for simulation
legacy_code ('sfcn_cmex_generate', def)
```

As a result the wrapper file appears:



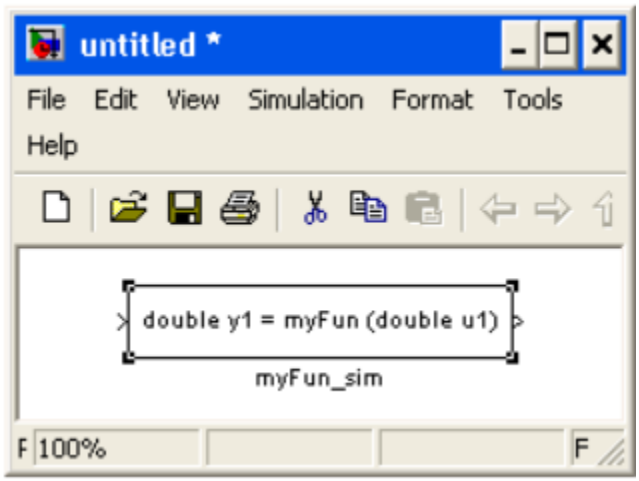
This needs to be compiled for Simulink

```
% create the mex file (compile the wrapper and the legacy code)
legacy_code ('compile', def)
```

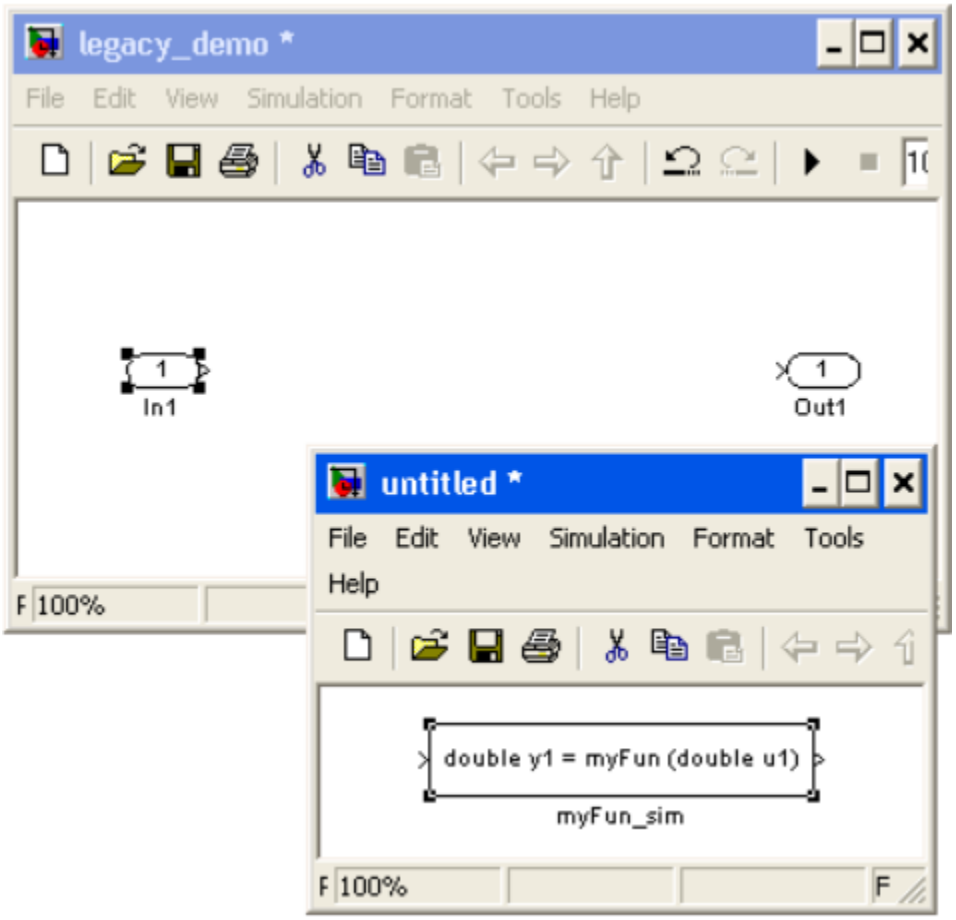



```
% generate Simulink block
% this will create a new model with a block linked to the new mex file
legacy_code ('slblock_generate', def)
```

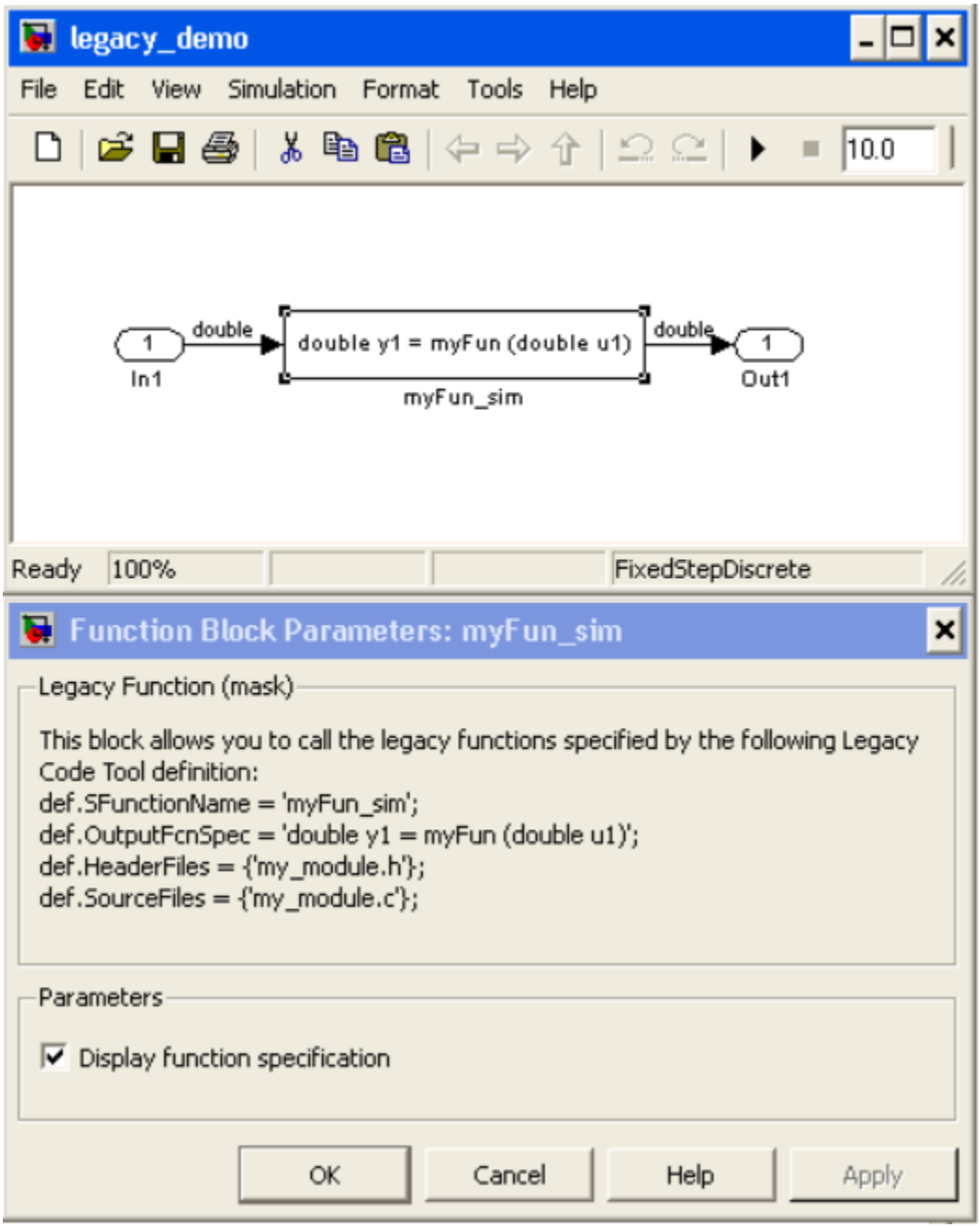
The last command in the template above will create a new model that contains the generated S-Function block.



Drag this block to the model where the S-Function was required.



The final model:



Note: Before executing QGen on the model, the variable `def` used to generate the S-Function block should be removed from the workspace as it may contain data in a cell format unsupported by QGen, causing an error during the export of the model to XMI.

After exporting the model to XMI and generating code we can see that the code contains call to the myFun function:

```
void legacy_demo_legacy_demo_comp
  (GAREAL In1,
   GAREAL *Out1)
{
```

(continues on next page)

(continued from previous page)

```
   GAREAL In1_out1;
   GAREAL myFun_sim_out1;

   /* Block legacy_demo/In1 */
   In1_out1 = In1;
   /* End Block legacy_demo/In1 */

   /* Block legacy_demo/myFun_sim */
   myFun_sim_out1 = myFun (In1_out1);
   /* End Block legacy_demo/myFun_sim */

   /* Block legacy_demo/Out1 */
   *Out1 = myFun_sim_out1;
   /* End Block legacy_demo/Out1 */

}
```

7.5.2 Function prototype syntax

The goal of the function prototype is to define a mapping between the block interface and function arguments/return values. Special identifier patterns are used to denote block ports and mask parameters

The syntax supported both by Simulink and `qgenc` contains the following elements:

```
[<return datatype> <return var>] <function name> (<arg1>, <arg2> ... <argN>)
```

where:

- `<return datatype>` is a valid Simulink data type
- `<return var>` and `<argX>` are references to inputs outputs or paramters in form
  - `u<no>` – input <no> where <no> in number of the input port
  - `y<no>` – output <no> where <no> is number of the output port
  - `p<no>` – parameter <no> where <no> is a parameter SParameter<no> defined in MaskVariables
- in case block input or output is an array then `<argX>` may refer to its size:
  - `size([u|y]<no>)` – length of the first dimension of input/output <no>
  - `size([u|y]<no>, 1)` – length of the first dimension of input/output <no>
  - `size([u|y]<no>, 2)` – length of the second dimension of input/output <no>

7.6 Option B: generate S-Function wrappers

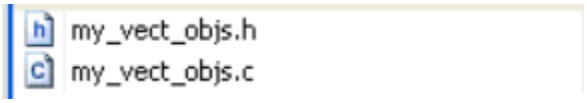
This option applies when the S-Function block in not generated using the legacy code tool (or one did not provide the full prototype specification). When function prototype specification is not found in the block specification then `qgenc` will generate a module named <modelname>_wrappers which contains a function for each generated block. The workflow is as follows:

- Run `qgenc` on a Simulink model to generate code
- `qgenc` will produce a module for each atomic subsystem + a special module containing function wrappers for each S-Function with no prototype.
- two files are generated for this special wrappers module:
  - `<modelname>_wrappers.[h|ads]` – the prototypes of wrapper functions
  - `<modelname>_wrappers.[c|adb].template` – empty function stubs for each wrapper function
- the wrappers module will contain a function for each S-Function block in form

  `<S-Function_name> (<inport 1>, <inport 2>, ..., <inport N>, <outport1>, <outport 2>, ..., <outport M>)`
- if this was the first time to generate code for this model then rename the file "<modelname>_wrappers.[c|adb].template" to "<modelname>_wrappers.[c|adb]" and write an appropriate function call in each wrapper function
- in case of repeated code generation simply check that the function calls in the wrapper module generated earlier are not changed (the h/ads file is always overwritten, c/adb file remains inteact as the new file created by `qgenc` has ".template" suffix)
- while compiling the code provide the legacy code module in include and lib paths

7.6.1 Example

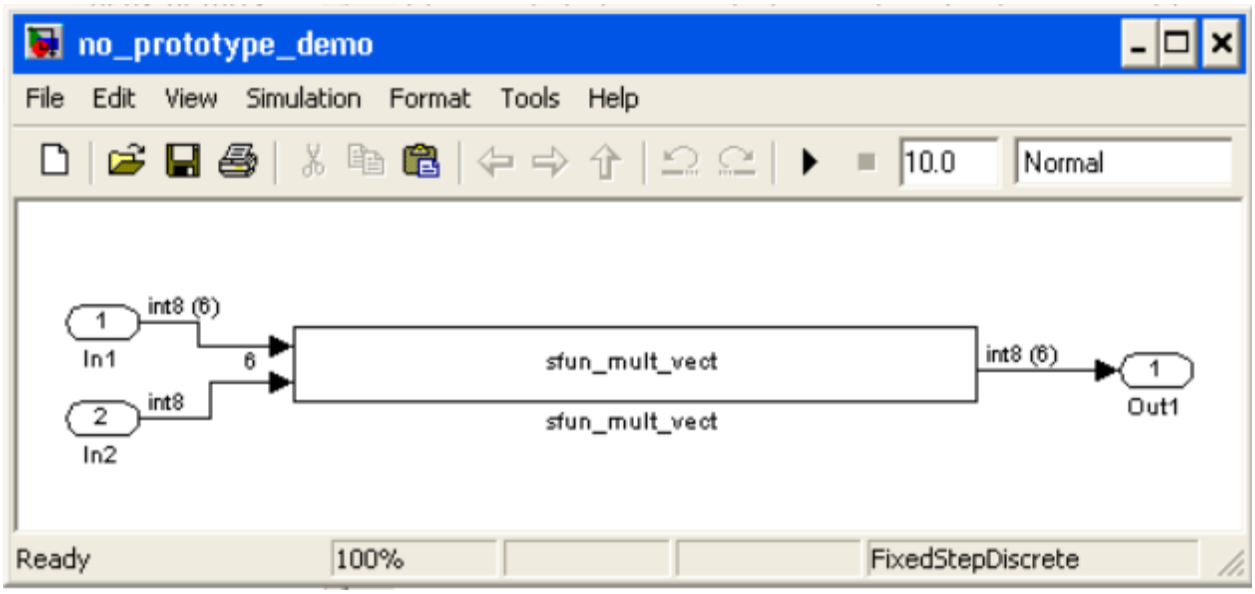
Let us consider a model mutlvect.mdl that calls a function mult_vect from library my_vect_ops multiplying a vector with constant.



Library my_vect_ops

my_vect_ops.h

```
#ifndef MY_VECT_OPS_H
#define MY_VECT_OPS_H

/**
 * Copies contents of vector to vector2 and multiplies each element
 * with multiplier.
**/
void mult_vect (int *vector, int vector_len, int multiplier, int *vector2);

#endif
```



Generated wrapper functions

The model contains a block named "sfun_mult_vect" with one input port of type int8[6], one input of type int8 and one output of type int8[6].

The wrappers generated by `qgenc` are as follows:

no_prototype_demo_wrappers.h

```
/* Copyright (C) Project P Consortium */
/*
 * @generated with GNAT Model Compiler 1.0w
 * Command line arguments:
 *   -l c no_prototype_demo.xmi
 *   --clean --pre-process-xmi
*/

#ifndef NO_PROTOTYPE_DEMO_WRAPPERS_H
#define NO_PROTOTYPE_DEMO_WRAPPERS_H
#include "qgen_types.h"

extern void no_prototype_demo_wrappers_sfun_mult_vect
  (GAINT8 In1_out1[6],
   GAINT8 In2_out1,
   GAINT8 sfun_mult_vect_out1[6]);

#endif
/*  @EOF  */
```

no_prototype_demo_wrappers.c.template

```
/* Copyright (C) Project P Consortium */
/*
 * @generated with GNAT Model Compiler 1.0w
```

(continues on next page)

(continued from previous page)

```
 * Command line arguments:
 *   -l c no_prototype_demo.xmi
 *   --clean --pre-process-xmi
*/

#include "no_prototype_demo_wrappers.h"

void no_prototype_demo_wrappers_sfun_mult_vect
  (GAINT8 In1_out1[6],
   GAINT8 In2_out1,
   GAINT8 sfun_mult_vect_out1[6])
{
}
/*  @EOF  */
```

Call to S-Function

no_prototype_demo.c

```
/* Copyright (C) Project P Consortium */
/*
 * @generated with GNAT Model Compiler 1.0w
 * Command line arguments:
 *   -l c no_prototype_demo.xmi
 *   --clean --pre-process-xmi
*/

#include "no_prototype_demo.h"

void no_prototype_demo_no_prototype_demo_init (void) {
}
void no_prototype_demo_no_prototype_demo_comp
  (GAINT8 In1[6],
   GAINT8 In2,
   GAINT8 Out1[6])
{
   GAINT8 In1_out1[6];
   GAINT8 In2_out1;
   GAINT8 sfun_mult_vect_out1[6];

   /* Block no_prototype_demo/In1 */
   for (GAUINT8 i = 0; i <= 5; i++) {
      In1_out1[i] = In1[i];
   }
   /* End Block no_prototype_demo/In1 */

   /* Block no_prototype_demo/In2 */
   In2_out1 = In2;
   /* End Block no_prototype_demo/In2 */

   /* Block no_prototype_demo/sfun_mult_vect */
```

(continues on next page)

(continued from previous page)

```
   (void) no_prototype_demo_wrappers_sfun_mult_vect (In1_out1, In2_out1, sfun_mult_vect_
↪out1);
   /* End Block no_prototype_demo/sfun_mult_vect */

   /* Block no_prototype_demo/Out1 */
   for (GAUINT8 i_1 = 0; i_1 <= 5; i_1++) {
      Out1[i_1] = sfun_mult_vect_out1[i_1];
   }
   /* End Block no_prototype_demo/Out1 */

}
/*  @EOF  */
```

Manual code

The code written manually inside of the generated multvect_template stubs:

no_prototype_demo_wrappers.c

```
/* Copyright (C) Project P Consortium */
/*
 * @generated with GNAT Model Compiler 1.0w
 * Command line arguments:
 *   -l c no_prototype_demo.xmi
 *   --clean --pre-process-xmi
*/

#include "no_prototype_demo_wrappers.h"
/* the legacy code specs */
#include "my_vect_ops.h"

void no_prototype_demo_wrappers_sfun_mult_vect
  (GAINT8 In1_out1[6],
   GAINT8 In2_out1,
   GAINT8 sfun_mult_vect_out1[6])
{
   /* call external function. At this point the vector length is
      statically known, so we pass the length argument as hard-coded
      constant
    */
    mult_vect (In1_out1, 6, In2_out1, sfun_mult_vect_out1);
}
/*  @EOF  */
```

From this point on, the no_prototype_demo_wrappers.c shall be made available at link time and it remains intact when new code is generated (unless the block interface changes).

7.7 Compiling and linking the generated code

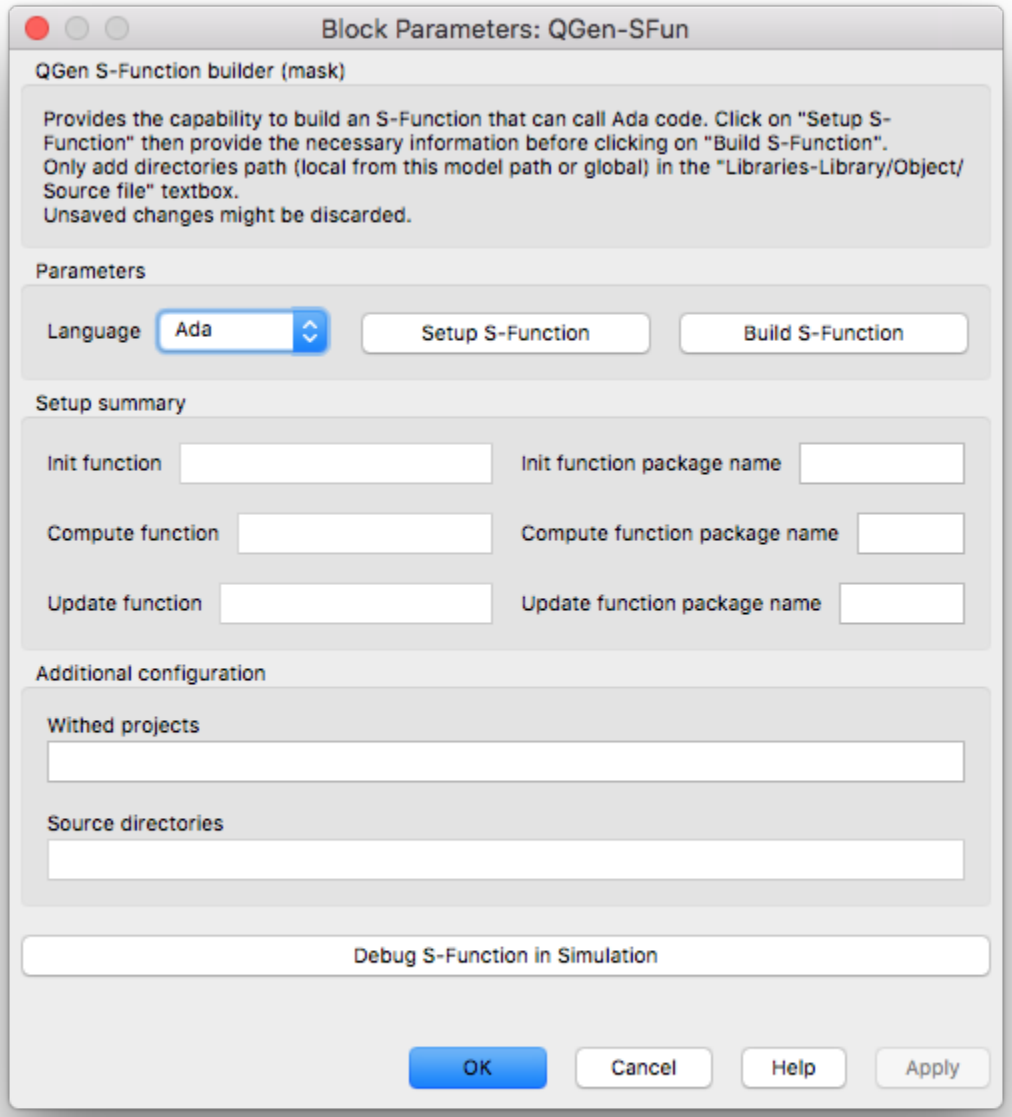
- The header file referenced in the HeaderFiles parameter shall be available at compile time.
- The source file referenced in SourceFiles shall be available to the linker

7.8 Calling C or Ada codebases using QGen-SFun block

The QGen-SFun block is available in the Simulink Library Browser under the QGen Toolset Library.

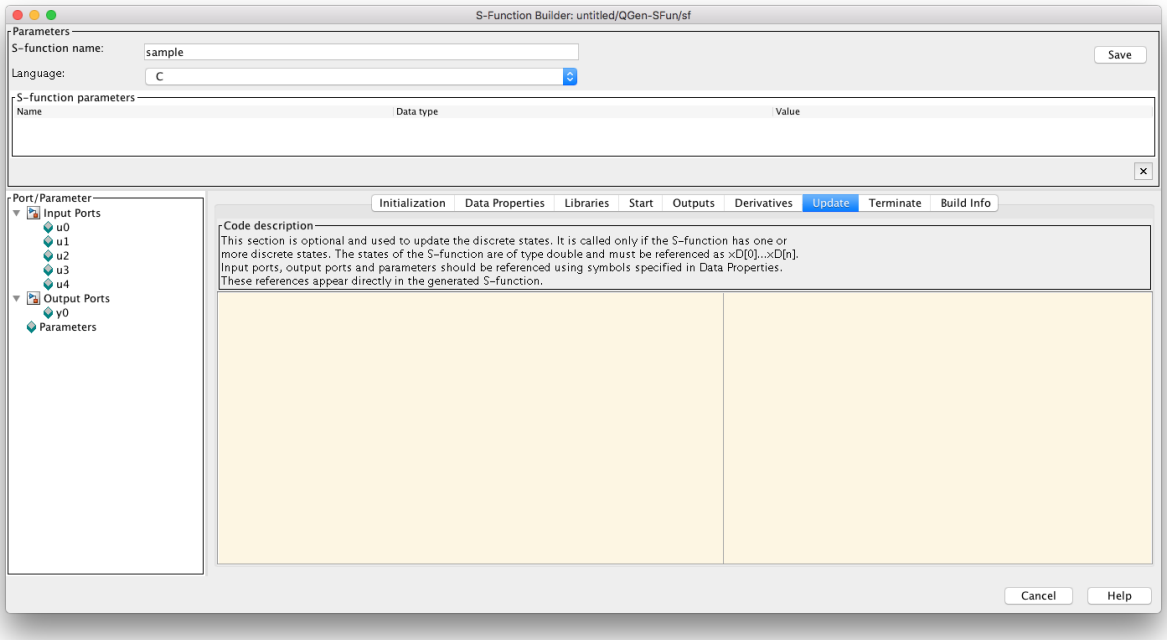
To use it simply drag it to the model from which you want to call your existing C or Ada code.

To configure the code, double click on the block. The following dialog will appear:



The first step is to choose the language that you are calling from the Language dropdown list.

Then, click on `Setup S-Function` and the S-Function builder panel documented here https://www.mathworks.com/help/simulink/sfg/s-function-builder-dialog-box.html will pop up.



Type the name of your S-Function binary within the `S-Function name` field.

Keep the language within that panel set to C, even if you chose Ada in the first screen. This language corresponds to the language called by Simulink and we need it to be C.

In the `Initialization` screen, set the number of discrete states and an array initializing them, note that each variable will be of type Float in Ada or single in C in the function prototypes.

In the `Data properties` screen, set the Input ports, output ports and data type attributes as needed. Changes to the parameters tab will not be taken into account.

In the `Libraries` screen, add the relative path to the sources directories of your code within the `Library/Object/Source files (one per line)` panel. Only paths to directories are supported by the QGen-SFun block.

The `External function declarations` only has to be filled when linking Ada code. When calling Ada you first have to make sure that you exported your Ada function(s) symbol(s) to C, as written in the example below:

```
package Ada_Sfun is
   procedure do_Compute(u0 : Long_Float;
                        u1 : Long_Float;
                        u2 : Long_Float;
                        u3 : Long_Float;
                        u4 : Long_Float;
                        y0 : out Long_Float);

   pragma Export (C, do_Compute,  "do_Compute");
end Ada_Sfun;
```

Warning: You have to Export your function with a symbol equal to its name.

Then you have to add in `External function declarations` the equivalent C prototypes for the exported symbol. Which, for the example above, would be :

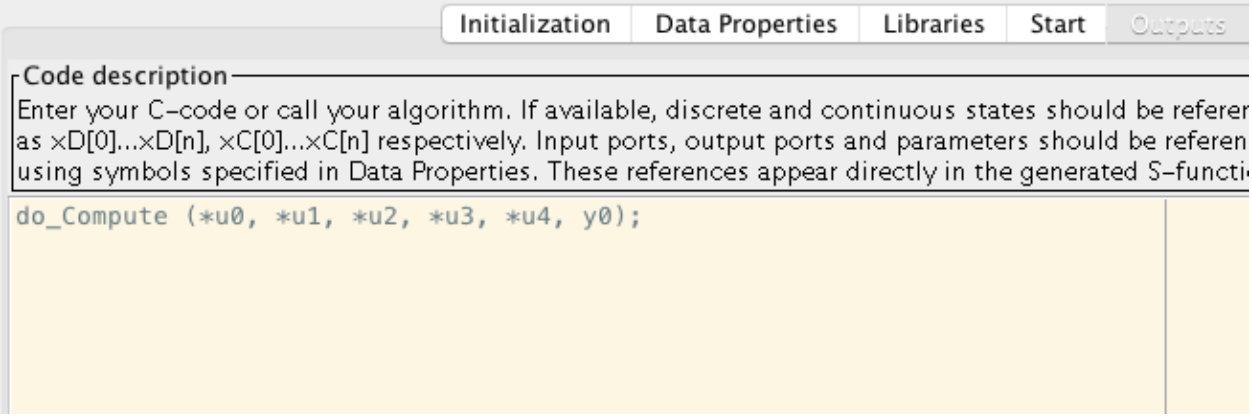
```
extern void do_Compute (const real_T u0,
                        const real_T u1,
                        const real_T u2,
                        const real_T u3,
                        const real_T u4,
                        const real_T * y0);
```

Be sure to realize this step for each function that you want to call directly from Simulink. In effect you will only have to do this for at most 3 functions, one for the `Outputs` call (mandatory) and two for `Start` and `Update` in the case where you have discrete states.

Finally the last step is to call the C symbol of your function(s) in the `Outputs` tab. If you have states you also should call the necessary functions in the `Start` and `Update` tab.

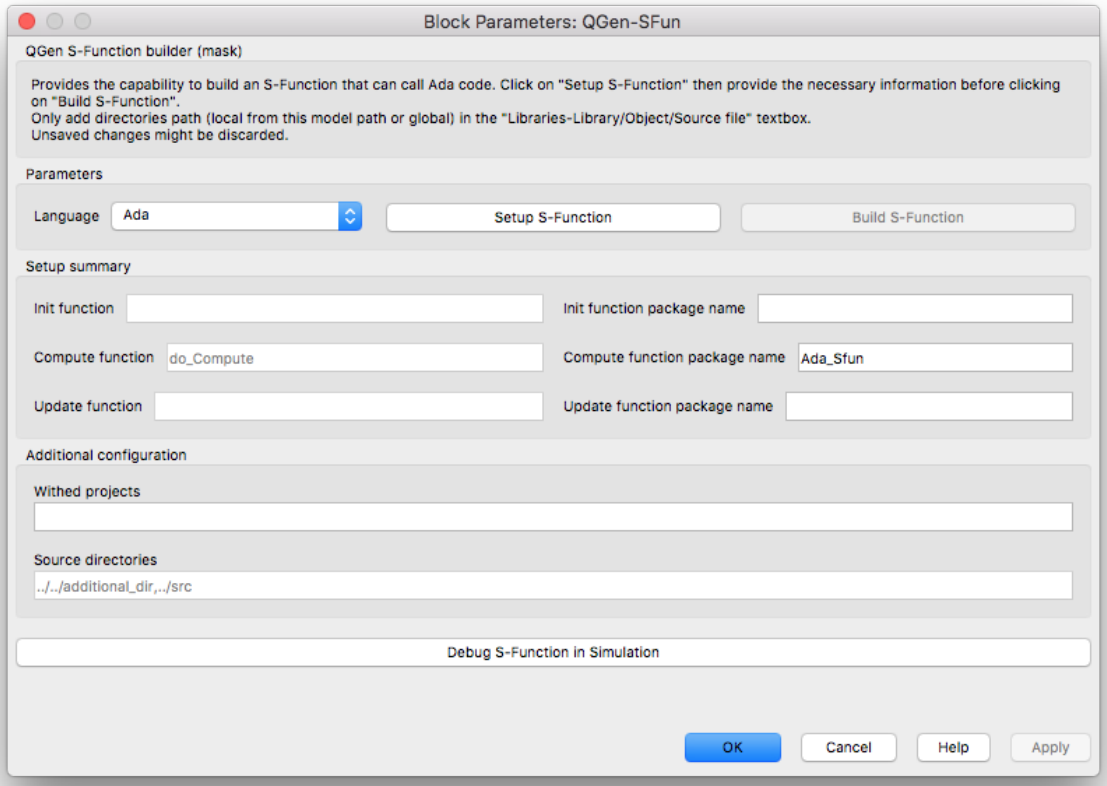
Warning: Do not type any extra C code in these panels. The only code that you should add is the call to your function. If you wish to realize additional processing of the inputs, you should do so within the called code, or call a wrapper that does the desired processing of the inputs.

In our example we would write the following call, using the names of the ports defined in `Data properties`. The states are passed as an array of single named `xD`.

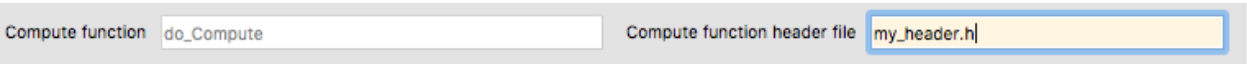


Warning: Pass the arguments to your function in the same order they were defined in `Data Properties`. Changing the order, omitting a parameter or combining parameters would cause the S-Function to be built correctly but the QGen code generation would not be correct. All ports and states are passed as pointers, if you are treating your input ports as scalar, make sure to dereference them, like in the example above.

Once the calls are complete click `Save` then `Close`. Some of your changes will appear in the `Setup summary` panel.



In C make sure to fill the header file field next to the functions that are called. This is mandatory for code generation.



After reviewing or completing the setup, click on `Build S-Function`. Your S-Function will be automatically built based on the provided setup. In the case of compilation errors, they will be detailed in the MATLAB command window.

Fix them by going back to `Setup S-Function` and editing the necessary fields. You can find the generated files within `[libname]_wrappers`.

When the build is sucessful you are ready to connect your block. You can also generate code for your model and QGen will call the correct functions.

Note: If your block is already connected and your reopen the `Setup S-Function` panel, connecting signals will be deleted. Simply reconnect them.

7.8.1 Debugging the QGen S-Function during simulation

After building your QGen S-Function, you can simply debug your code during the Simulink Simulation by clicking on `Debug S-Function in Simulation`.

This will spawn a GNAT Studio or GPS instance that will automatically attach to Simulink. When the Debugger Console is available, you can start or step within the simulation and the {GNAT Studio|GPS} Debugger will automatically break when reaching your S-Function call.

Note: As soon as the S-Function code breakpoint is reached, MATLAB will hang. This is normal, just continue the code execution after debugging this step to unblock MATLAB. Once MATLAB is unblocked, you can step, continue or stop the simulation.

All standard debugging capabilities are available after your stepped within the S-Function code.

CHAPTER

EIGHT

QGEN CONSTRAINTS ON INPUT MODELS

8.1 Simulink Version

The QGen toolset is compatible with MATLAB and Simulink versions R2015b and newer.

8.2 Simulink Block Types and Constraints

This section lists the Simulink block types supported by QGen and details the constraints under which they are supported. This information is organised in the following sections:

- *Supported Simulink Block Types (alphabetically)*
- *Supported Simulink Block Types (by library)*
- *Simulink Block Constraints*

8.2.1 Supported Simulink Block Types (alphabetically)

	Block Type	Library	Constraints
1			
	1-D Lookup Table	*Lookup Tables*	*1-D Lookup Table Constraints*
2			
	2-D Lookup Table	*Lookup Tables*	*2-D Lookup Table Constraints*
A			
	ASCII to String	*String*	None
	Abs	*Math Operations*	None
	Action Port	*Ports and Subsystems*	None
	Assertion	*Model Verification*	*Assertion Constraints*
	Assignment	*Math Operations*	*Assignment Constraints*
	Atomic Subsystem	*Ports and Subsystems*	None
B			
	Backlash	*Discontinuities*	*Backlash Constraints*
	Bias	*Math Operations*	None
	Bitwise Operator	*Logic and Bit Operations*	None
	Bus Assignment	*Signal Routing*	None
	Bus Creator	*Signal Routing*	*Bus Creator Constraints*
	Bus Selector	*Signal Routing*	None
	Bus to Vector	*Signal Attributes*	None

continues on next page

Table 1 – continued from previous page

	Block Type	Library	Constraints
C			
	C Caller	*User-Defined Functions*	None
	Chart	*Stateflow*	None
	Clock	*Sources*	None
	CodeReuse Subsystem	*Ports and Subsystems*	None
	Combinatorial Logic	*Logic and Bit Operations*	None
	Compare To Constant	*Logic and Bit Operations*	*Compare To Constant Constraints*
	Compare To Zero	*Logic and Bit Operations*	*Compare To Zero Constraints*
	Complex to Real-Imag	*Math Operations*	None
	Compose String	*String*	*Compose String Constraints*
	Constant	*Sources*	None
D			
	Data Store Memory	*Signal Routing*	None
	Data Store Read	*Signal Routing*	None
	Data Store Write	*Signal Routing*	None
	Data Type Conversion	*Signal Attributes*	*Data Type Conversion Constraints*
	Data Type Duplicate	*Signal Attributes*	*Data Type Duplicate Constraints*
	Dead Zone	*Discontinuities*	None
	Delay	*Discrete*	*Delay Constraints*
	Demux	*Signal Routing*	*Demux Constraints*
	Detect Change	*Logic and Bit Operations*	*Detect Change Constraints*
	Detect Decrease	*Logic and Bit Operations*	*Detect Decrease Constraints*
	Detect Increase	*Logic and Bit Operations*	*Detect Increase Constraints*
	Difference	*Discrete*	*Difference Constraints*
	Digital Clock	*Sources*	None
	Direct Lookup Table (n-D)	*Lookup Tables*	*Direct Lookup Table (n-D) Constraints*
	Discrete Derivative	*Discrete*	*Discrete Derivative Constraints*
	Discrete State-Space	*Discrete*	*Discrete State-Space Constraints*
	Discrete Transfer Fcn	*Discrete*	*Discrete Transfer Fcn Constraints*
	Discrete-Time Integrator	*Discrete*	*Discrete-Time Integrator Constraints*
	Display	*Sinks*	*Display Constraints*
	DocBlock	*Model-Wide Utilities*	*DocBlock Constraints*
E			
	Enable	*Ports and Subsystems*	*Enable Constraints*
	Enabled Subsystem	*Ports and Subsystems*	None
	Enabled and Triggered Subsystem	*Ports and Subsystems*	*Enabled and Triggered Subsystem Constraints*
	Enumerated Constant	*Sources*	None
	Extract Bits	*Logic and Bit Operations*	*Extract Bits Constraints*
F			
	Fcn	*User-Defined Functions*	None
	For Each	*Ports and Subsystems*	*For Each Constraints*
	For Each Subsystem	*Ports and Subsystems*	*For Each Subsystem Constraints*
	For Iterator	*Ports and Subsystems*	*For Iterator Constraints*
	For Iterator Subsystem	*Ports and Subsystems*	None
	From	*Signal Routing*	None

continues on next page

Table 1 – continued from previous page

	Block Type	Library	Constraints
	Function-Call Generator	*Ports and Subsystems*	*Function-Call Generator Constraints*
G			
	Gain	*Math Operations*	None
	Goto	*Signal Routing*	None
	Goto Tag Visibility	*Signal Routing*	None
	Ground	*Sources*	None
H			
	Hit Crossing	*Discontinuities*	*Hit Crossing Constraints*
I			
	IC	*Signal Attributes*	None
	If	*Ports and Subsystems*	*If Constraints*
	If Action Subsystem	*Ports and Subsystems*	None
	Inport	*Sources*	*Inport Constraints*
	InportShadow	*Sources*	None
	Interpolation Using Prelookup	*Lookup Tables*	*Interpolation Using Prelookup Constraints*
L			
	Logical Operator	*Logic and Bit Operations*	None
	Lookup	*Lookup Tables*	*Lookup Constraints*
	Lookup Table (2-D)	*Lookup Tables*	*Lookup Table (2-D) Constraints*
	Lookup Table Dynamic	*Lookup Tables*	*Lookup Table Dynamic Constraints*
M			
	MATLAB Function	*User-Defined Functions*	*MATLAB Function Constraints*
	Math Function	*Math Operations*	*Math Function Constraints*
	Matrix Concatenate	*Math Operations*	None
	Memory	*Discrete*	None
	Merge	*Signal Routing*	*Merge Constraints*
	MinMax	*Math Operations*	None
	Model	*Ports and Subsystems*	None
	Model Info	*Model-Wide Utilities*	*Model Info Constraints*
	Model Variants	*Ports and Subsystems*	*Model Variants Constraints*
	Multiport Switch	*Signal Routing*	*Multiport Switch Constraints*
	Mux	*Signal Routing*	*Mux Constraints*
O			
	Outport	*Ports and Subsystems*	*Outport Constraints*
P			
	Permute Dimensions	*Math Operations*	None
	Prelookup	*Lookup Tables*	None
	Product	*Math Operations*	*Product Constraints*
Q			
	QGen S-Function builder	*QGen*	None
R			
	Rate Limiter	*Discontinuities*	None
	Rate Transition	*Signal Attributes*	None
	Real-Imag to Complex	*Math Operations*	None
	Relational Operator	*Logic and Bit Operations*	None
	Relay	*Discontinuities*	None

continues on next page

Table 1 – continued from previous page

	Block Type	Library	Constraints
	Reshape	*Math Operations*	None
	Rounding Function	*Math Operations*	None
S			
	S-Function	*User-Defined Functions*	*S-Function Constraints*
	Saturation	*Discontinuities*	None
	Saturation Dynamic	*Discontinuities*	*Saturation Dynamic Constraints*
	Scope	*Sinks*	*Scope Constraints*
	Selector	*Signal Routing*	*Selector Constraints*
	Shift Arithmetic	*Logic and Bit Operations*	*Shift Arithmetic Constraints*
	Sign	*Math Operations*	None
	Signal Conversion	*Signal Attributes*	None
	Signal Specification	*Signal Attributes*	*Signal Specification Constraints*
	Signal Viewer Scope	*Sinks*	*Signal Viewer Scope Constraints*
	Sqrt	*Math Operations*	None
	State Transition Table	*Stateflow*	None
	String Constant	*String*	*String Constant Constraints*
	String Length	*String*	None
	String to ASCII	*String*	None
	Subsystem	*Ports and Subsystems*	None
	Sum	*Math Operations*	*Sum Constraints*
	Switch	*Signal Routing*	None
	Switch Case	*Ports and Subsystems*	None
	Switch Case Action Subsystem	*Ports and Subsystems*	None
T			
	Tapped Delay	*Discrete*	None
	Terminator	*Sinks*	*Terminator Constraints*
	To Workspace	*Sinks*	*To Workspace Constraints*
	Transfer Fcn First Order	*Discrete*	*Transfer Fcn First Order Constraints*
	Transfer Fcn Lead or Lag	*Discrete*	*Transfer Fcn Lead or Lag Constraints*
	Transfer Fcn Real Zero	*Discrete*	*Transfer Fcn Real Zero Constraints*
	Trigger	*Ports and Subsystems*	*Trigger Constraints*
	Triggered Subsystem	*Ports and Subsystems*	*Triggered Subsystem Constraints*
	Trigonometric Function	*Math Operations*	*Trigonometric Function Constraints*
	Truth Table	*Stateflow*	None
U			
	Unary Minus	*Math Operations*	None
	Uniform Random Number	*Sources*	*Uniform Random Number Constraints*
	Unit Delay	*Discrete*	None
	UnitConversion	*Signal Attributes*	*UnitConversion Constraints*
V			
	Vector Concatenate	*Math Operations*	None
W			
	While Iterator	*Ports and Subsystems*	None
	While Iterator Subsystem	*Ports and Subsystems*	None

continues on next page

Table 1 – continued from previous page

	Block Type	Library	Constraints
	Width	*Signal Attributes*	None
	Wrap To Zero	*Discontinuities*	*Wrap To Zero Constraints*
Z			
	Zero-Order Hold	*Discrete*	None
n			
	n-D Lookup Table	*Lookup Tables*	*n-D Lookup Table Constraints*

8.2.2 Supported Simulink Block Types (by library)

QGen

Block Type	Constraints
QGen S-Function builder	None

Discontinuities

Link to MathWorks documentation: Discontinuities

Block Type	Constraints
Backlash	*Backlash Constraints*
Dead Zone	None
Hit Crossing	*Hit Crossing Constraints*
Rate Limiter	None
Relay	None
Saturation	None
Saturation Dynamic	*Saturation Dynamic Constraints*
Wrap To Zero	*Wrap To Zero Constraints*

Discrete

Link to MathWorks documentation: Discrete

Block Type	Constraints
Delay	*Delay Constraints*
Difference	*Difference Constraints*
Discrete Derivative	*Discrete Derivative Constraints*
Discrete State-Space	*Discrete State-Space Constraints*
Discrete Transfer Fcn	*Discrete Transfer Fcn Constraints*
Discrete-Time Integrator	*Discrete-Time Integrator Constraints*
Memory	None
Tapped Delay	None
Transfer Fcn First Order	*Transfer Fcn First Order Constraints*
Transfer Fcn Lead or Lag	*Transfer Fcn Lead or Lag Constraints*
Transfer Fcn Real Zero	*Transfer Fcn Real Zero Constraints*
Unit Delay	None
Zero-Order Hold	None

Logic and Bit Operations

Link to MathWorks documentation: Logic and Bit Operations

Block Type	Constraints
Bitwise Operator	None
Combinatorial Logic	None
Compare To Constant	*Compare To Constant Constraints*
Compare To Zero	*Compare To Zero Constraints*
Detect Change	*Detect Change Constraints*
Detect Decrease	*Detect Decrease Constraints*
Detect Increase	*Detect Increase Constraints*
Extract Bits	*Extract Bits Constraints*
Logical Operator	None
Relational Operator	None
Shift Arithmetic	*Shift Arithmetic Constraints*

Lookup Tables

Link to MathWorks documentation: Lookup Tables

Block Type	Constraints
1-D Lookup Table	*1-D Lookup Table Constraints*
2-D Lookup Table	*2-D Lookup Table Constraints*
Direct Lookup Table (n-D)	*Direct Lookup Table (n-D) Constraints*
Interpolation Using Prelookup	*Interpolation Using Prelookup Constraints*
Lookup	*Lookup Constraints*
Lookup Table (2-D)	*Lookup Table (2-D) Constraints*
Lookup Table Dynamic	*Lookup Table Dynamic Constraints*
Prelookup	None
n-D Lookup Table	*n-D Lookup Table Constraints*

Math Operations

Link to MathWorks documentation: Math Operations

Block Type	Constraints
Abs	None
Assignment	*Assignment Constraints*
Bias	None
Complex to Real-Imag	None
Gain	None
Math Function	*Math Function Constraints*
Matrix Concatenate	None
MinMax	None
Permute Dimensions	None
Product	*Product Constraints*
Real-Imag to Complex	None
Reshape	None
Rounding Function	None
Sign	None
Sqrt	None
Sum	*Sum Constraints*
Trigonometric Function	*Trigonometric Function Constraints*
Unary Minus	None
Vector Concatenate	None

Model Verification

Link to MathWorks documentation: Model Verification

Block Type	Constraints
Assertion	*Assertion Constraints*

Ports and Subsystems

Link to MathWorks documentation: Ports and Subsystems

Block Type	Constraints
Action Port	None
Atomic Subsystem	None
CodeReuse Subsystem	None
Enable	*Enable Constraints*
Enabled Subsystem	None
Enabled and Triggered Subsystem	*Enabled and Triggered Subsystem Constraints*
For Each	*For Each Constraints*
For Each Subsystem	*For Each Subsystem Constraints*
For Iterator	*For Iterator Constraints*
For Iterator Subsystem	None
Function-Call Generator	*Function-Call Generator Constraints*
If	*If Constraints*
If Action Subsystem	None
Inport	*Inport Constraints*
InportShadow	None
Model	None
Model Variants	*Model Variants Constraints*
Outport	*Outport Constraints*
Subsystem	None
Switch Case	None
Switch Case Action Subsystem	None
Trigger	*Trigger Constraints*
Triggered Subsystem	*Triggered Subsystem Constraints*
While Iterator	None
While Iterator Subsystem	None

Signal Attributes

Link to MathWorks documentation: Signal Attributes

Block Type	Constraints
Bus to Vector	None
Data Type Conversion	*Data Type Conversion Constraints*
Data Type Duplicate	*Data Type Duplicate Constraints*
IC	None
Rate Transition	None
Signal Conversion	None
Signal Specification	*Signal Specification Constraints*
UnitConversion	*UnitConversion Constraints*
Width	None

Signal Routing

Link to MathWorks documentation: Signal Routing

Block Type	Constraints
Bus Assignment	None
Bus Creator	*Bus Creator Constraints*
Bus Selector	None
Data Store Memory	None
Data Store Read	None
Data Store Write	None
Demux	*Demux Constraints*
From	None
Goto	None
Goto Tag Visibility	None
Merge	*Merge Constraints*
Multiport Switch	*Multiport Switch Constraints*
Mux	*Mux Constraints*
Selector	*Selector Constraints*
Switch	None
Vector Concatenate	None

Sinks

Link to MathWorks documentation: Sinks

Block Type	Constraints
Display	*Display Constraints*
Outport	*Outport Constraints*
Scope	*Scope Constraints*
Signal Viewer Scope	*Signal Viewer Scope Constraints*
Terminator	*Terminator Constraints*
To Workspace	*To Workspace Constraints*

Sources

Link to MathWorks documentation: Sources

Block Type	Constraints
Clock	None
Constant	None
Digital Clock	None
Enumerated Constant	None
Ground	None
Inport	*Inport Constraints*
InportShadow	None
Uniform Random Number	*Uniform Random Number Constraints*

User-Defined Functions

Link to MathWorks documentation: User-Defined Functions

Block Type	Constraints
C Caller	None
Fcn	None
MATLAB Function	*MATLAB Function Constraints*
S-Function	*S-Function Constraints*

Model-Wide Utilities

Link to MathWorks documentation: Model-Wide Utilities

Block Type	Constraints
DocBlock	*DocBlock Constraints*
Model Info	*Model Info Constraints*

String

Link to MathWorks documentation: String

Block Type	Constraints
ASCII to String	None
Compose String	*Compose String Constraints*
String Constant	*String Constant Constraints*
String Length	None
String to ASCII	None

Stateflow

Block Type	Constraints
Chart	None
State Transition Table	None
Truth Table	None

8.2.3 Simulink Block Constraints

1-D Lookup Table

Links to Mathworks Documentation

Block Type	Library
1-D Lookup Table	Lookup Tables

QGen Usage Constraints

- Supported interpolation methods

  Flat and Linear interpolation methods are supported. Nearest and Cubic spline methods are not supported.

- Supported extrapolation methods

  Clip and Linear extrapolation methods are supported. Cubic spline method is not supported.

- Maximum number of table dimensions

  Table dimensions higher than 2 are not supported.

2-D Lookup Table

Links to Mathworks Documentation

Block Type	Library
2-D Lookup Table	Lookup Tables

QGen Usage Constraints

- Supported interpolation methods

  Flat and Linear interpolation methods are supported. Nearest and Cubic spline methods are not supported.

- Supported extrapolation methods

  Clip and Linear extrapolation methods are supported. Cubic spline method is not supported.

- Maximum number of table dimensions

  Table dimensions higher than 2 are not supported.

Assertion

Links to Mathworks Documentation

Block Type	Library
Assertion	Model Verification

QGen Usage Constraints

- Empty Callback

  Using a callback expression for this block is not supported by QGen. The callback expression must be empty.

Assignment

Links to Mathworks Documentation

Block Type	Library
Assignment	Math Operations

QGen Usage Constraints

- Number of Output Dimensions

  The "Number of output dimensions" parameter should be 1 or 2 as QGen does not support data with larger dimensionality.

- No Partial Assignment

  When the "Initialize output (Y)" parameter is set to "Specify size for each dimension in table", the "Action if any output element is not assigned" parameter should be set to "Error".

Backlash

Links to Mathworks Documentation

Block Type	Library
Backlash	Discontinuities

QGen Usage Constraints

- Supported Size

  For non-scalar values, the size of each dimension of the "Deadband width" and "Initial output" parameters should be equal to the size of the corresponding input data dimension.

Bus Creator

Links to Mathworks Documentation

Block Type	Library
Bus Creator	Signal Routing

QGen Usage Constraints

- Bus objects are not supported for virtual buses

  Bus objects are not supported for virtual buses. When creating a virtual bus make sure the BusObject / Output Data Type field is empty.

- Names of incoming signals must match the names in the bus object

  Bus objects are mandatory for non-virtual buses and the names of the incoming signals must match the names defined in the bus object.

Compare To Constant

Links to Mathworks Documentation

Block Type	Library
Compare To Constant	Logic and Bit Operations

QGen Usage Constraints

- Supported Size

  For non-scalar values, the size of each dimension of the parameter should be equal to the size of the corresponding input data dimension.

Compare To Zero

Links to Mathworks Documentation

Block Type	Library
Compare To Zero	Logic and Bit Operations

QGen Usage Constraints

- [Zero-crossing detection]

  The value of the Enable zero-crossing detection parameter is ignored by QGen as QGen only supports fixed-step solvers.

Compose String

Links to Mathworks Documentation

Block Type	Library
Compose String	String

QGen Usage Constraints

- Bounded string types only

  The output data type must be a bounded string, defined by stringtype(x)

- Exponential notation difference in Windows

  The implementation of ComposeString in QGen is using snprintf function from the stdio library. A known difference that occurs in Windows is that the exponential notation prints three decimal digits from generated C code (compiled with gcc) and Simulink simulation outputs two decimal digits.

Data Type Conversion

Links to Mathworks Documentation

Block Type	Library
Data Type Conversion	Signal Attributes

QGen Usage Constraints

- Integer rounding mode

  When MISRA violations are treated as errors, integer rounding mode must be "Zero".

Data Type Duplicate

Links to Mathworks Documentation

Block Type	Library
Data Type Duplicate	Signal Attributes

QGen Usage Constraints

- NotGenerated

  QGen supports models using this block to provide a seamless experience in Simulink, however no code is generated for this block. The code generated for the model is not affected.

Delay

Links to Mathworks Documentation

Block Type	Library
Delay	Discrete

QGen Usage Constraints

- Delay length source

  Delay length source must be set to “Dialog”

- External Reset

  External reset is not supported.

Demux

Links to Mathworks Documentation

Block Type	Library
Demux	Signal Routing

QGen Usage Constraints

- Input

  Bus signals should not be used at the input of this block.

  Note: Use BusCreator and BusSelector blocks instead of Mux and Demux to process bus signals, as recommended in the Simulink documentation.

Detect Change

Links to Mathworks Documentation

Block Type	Library
Detect Change	Logic and Bit Operations

QGen Usage Constraints

- Frame-based input processing not supported

  Input processing mode “Columns as channels (frame based)” is not supported.

Detect Decrease

Links to Mathworks Documentation

Block Type	Library
Detect Decrease	Logic and Bit Operations

QGen Usage Constraints

- Frame-based input processing not supported

  Input processing mode “Columns as channels (frame based)” is not supported.

Detect Increase

Links to Mathworks Documentation

Block Type	Library
Detect Increase	Logic and Bit Operations

QGen Usage Constraints

- Frame-based input processing not supported

  Input processing mode “Columns as channels (frame based)” is not supported.

Difference

Links to Mathworks Documentation

Block Type	Library
Difference	Discrete

QGen Usage Constraints

- Unsupported mode ‘Columns as channels’

  Parameter InputProcessing value ‘Columns as channels’ sets the block to frame-based input processing. This mode is not supported.

- Boolean input is not supported

  Simulink does not recommend to use this block with boolean types. A block with boolean input is rejected.

Direct Lookup Table (n-D)

Links to Mathworks Documentation

Block Type	Library
Direct Lookup Table (n-D)	Lookup Tables

QGen Usage Constraints

- Maximum number of table dimensions

  Maximum number of table dimensions is 2.

Discrete Derivative

Links to Mathworks Documentation

Block Type	Library
Discrete Derivative	Discrete

QGen Usage Constraints

- Matching input/output base types

  Base (element) types of input and output must be the same (follows from MISRA AC SLSF 002 Ver 1.0 2009, Section 3.1.2)

Discrete State-Space

Links to Mathworks Documentation

Block Type	Library
Discrete State-Space	Discrete

QGen Usage Constraints

- Parameters Value

  Parameters cannot be specified empty with []

  Note: Use zeros(..) instead of [] to specify an empty parameter

Discrete Transfer Fcn

Links to Mathworks Documentation

Block Type	Library
Discrete Transfer Fcn	Discrete

QGen Usage Constraints

- External Reset

  External reset is not supported.

- Input Data Type

  Only single or double data types are accepted in input. Integers are not accepted.

- Initial States

  The value of the initial states should be set to zero.

- Input Processing

  Input Processing parameter should be set to 'Elements as channels (sample based)'.

- Numerator

  Numerator parameter has to be set to Dialog and must be scalar or vector.

- Denominator

  Denominator parameter has to be set to Dialog and must be scalar or vector.

- Initial states

  Initial states has to be set to Dialog

- Scalar signals

  The input, output and InitialStates parameter must be scalar.

Discrete-Time Integrator

Links to Mathworks Documentation

Block Type	Library
Discrete-Time Integrator	Discrete

QGen Usage Constraints

- Integration Method

  Only the Forward Euler Integration and the Backward Euler Integration methods are supported.

- Show State Port

  Show state port is not supported.

- Ignore Limit

  Ignore limit and reset when linearizing is not supported.

- Conditionally activated subsystems

  The DTI cannot be called within a conditionally activated subsystem, such as enabled subsystems or function-call triggered subsystems.

- Multiple calls

  A DTI block cannot be called by multiple callers or multiple times by a single caller in the same time instance within function-call subsystems.

Display

Links to Mathworks Documentation

Block Type	Library
Display	Sinks

QGen Usage Constraints

- NotGenerated

  QGen supports models using this block to provide a seamless experience in Simulink, however no code is generated for this block. The code generated for the model is not affected.

DocBlock

Links to Mathworks Documentation

Block Type	Library
DocBlock	Model-Wide Utilities

QGen Usage Constraints

- NotGenerated

  QGen supports models using this block to provide a seamless experience in Simulink, however no code is generated for this block. The code generated for the model is not affected.

Enable

Links to Mathworks Documentation

Block Type	Library
Enable	Ports and Subsystems

QGen Usage Constraints

- Enable blocks are not supported in a top-level diagram

  An Enable block is not supported in a top-level diagram. A workaround is to wrap the contents of the top-level diagram into an Enabled Subsystem and pass the model-level signal that is designated for enabling the model's contents through a data port.

Enabled and Triggered Subsystem

Links to Mathworks Documentation

Block Type	Library
Enabled and Triggered Subsystem	Ports and Subsystems

QGen Usage Constraints

- Non-scalar input to trigger port is not supported

  Non-scalar input to trigger port is not supported.

Extract Bits

Links to Mathworks Documentation

Block Type	Library
Extract Bits	Logic and Bit Operations

QGen Usage Constraints

- Supported input data type

  The supported data types in input are: boolean and unsigned integers of length 8, 16 or 32 bits, single and double. Signed integers are not supported. Single and double data go through the block unchanged.

- Output data type

  An arbitrary number of bits to extract may be specified (between 1 and the input data length). If that number is not 8, 16 or 32, or if the "Preserve fixed-point scaling" mode has been chosen, Simulink may generate a fixed-point data type in output. This output data type will be allowed by QGen to be exported and later on converted to an integer (preserving sign and scaling) with a warning. At the model level, the user should enforce the data type conversion of the output to an integer capable of containing the output (preserving sign and scaling) using a Data Type Conversion block, in order to avoid the propagation of the fixed-point data type to the model.

- Output Fixed-Point Format

  The supported fixed-point data types accepted by QGen as output of this block in the case they are automatically generated have the following format: "sfix{M}_E{N}" or "ufix{M}_E{N}", where M is the number of bits extracted and N is the offset from the Least Significant Bit.

For Each

Links to Mathworks Documentation

Block Type	Library
For Each	Ports and Subsystems

QGen Usage Constraints

- No nested state

  A For Each Subsystem cannot contain blocks with state (e.g. Unit Delay).

- Parameter Partition

  Applying the ForEach operation on mask parameters of the containing subsystem instead of applying it to the subsystem inputs is not supported by QGen.

- Supported Dimensions

  "Partition Dimension" and "Concatenation Dimension" parameters should be 1 or 2 because QGen only supports 2-D data.

For Each Subsystem

Links to Mathworks Documentation

Block Type	Library
For Each Subsystem	Ports and Subsystems

QGen Usage Constraints

- No nested blocks with state

  A For Each Subsystem cannot contain blocks with state (e.g. Unit Delay).

For Iterator

Links to Mathworks Documentation

Block Type	Library
For Iterator	Ports and Subsystems

QGen Usage Constraints

- External Iteration Limit

  Input ports (External value for iterator variable or External value for iteration limit) must be connected to an interface port.

Function-Call Generator

Links to Mathworks Documentation

Block Type	Library
Function-Call Generator	Ports and Subsystems

QGen Usage Constraints

- Number of iterations must have a scalar value

  The value in the 'Number of Iterations' field must be a scalar

Hit Crossing

Links to Mathworks Documentation

Block Type	Library
Hit Crossing	Discontinuities

QGen Usage Constraints

- Hit crossing offset dimensions

  If the "Hit crossing offset" is non-scalar, it must have the same dimensions as the block's input.

If

Links to Mathworks Documentation

Block Type	Library
If	Ports and Subsystems

QGen Usage Constraints

- Maximum Number Of Input Ports

  The value of the "Number of inputs" parameter should be less or equal to 1024 as QGen does not support more input ports.

Inport

Links to Mathworks Documentation

Block Type	Library
Inport	Sources

QGen Usage Constraints

- "Latch input by delaying outside signal" must be off

  "Latch input by delaying outside signal" must be turned off.

- "Latch input for feedback signals" must be off

  "Latch input for feedback signals of function-call subsystem outputs" ("Latch input by copying inside signal" in earlier Simulink versions) must be turned off.

  Note: This parameter exists since MATLAB R2011a

- "Latch input by copying inside signal" must be off

  "Latch input by copying inside signal" must be turned off.

  Note: This parameter exists until MATLAB R2010b

Interpolation Using Prelookup

Links to Mathworks Documentation

Block Type	Library
Interpolation Using Prelookup	Lookup Tables

QGen Usage Constraints

- Max two dimensions

  The block supports up to two dimensions. The value of parameter "Table" shall be in either a vector or matrix. "NumberOrTableDimensions" shall be 1 or 2

- No explicitly selected dimensions

  The parameter NumSelectionDims shall be empty

- No saturation

  The value of SaturateOnIntegerOverflow parameter shall be either empty or "off"

Lookup

Links to Mathworks Documentation

Block Type	Library
Lookup	Lookup Tables

QGen Usage Constraints

- Supported Simulink version

  The maximum Simulink version supported for this block is R2010b. Use 1-D Lookup table instead from R2011a.

Lookup Table (2-D)

Links to Mathworks Documentation

Block Type	Library
Lookup Table (2-D)	Lookup Tables

QGen Usage Constraints

- Supported interpolation methods

  Flat and Linear interpolation methods are supported. Nearest and Cubic spline methods are not supported.

- Supported exrapolation methods

  Clip and Linear extrapolation methods are supported. Cubic spline method is not supported.

- Maximum number of table dimensions

  Table dimensions higher than 2 are not supported.

Lookup Table Dynamic

Links to Mathworks Documentation

Block Type	Library
Lookup Table Dynamic	Lookup Tables

QGen Usage Constraints

- No Saturation

  The value of DoSatur parameter shall be either empty or "off"

MATLAB Function

Links to Mathworks Documentation

Block Type	Library
MATLAB Function	User-Defined Functions

QGen Usage Constraints

- Generates a wrapper only

  MATLAB functions will only generate a call to a wrapper that needs to be completed by the user.

Math Function

Links to Mathworks Documentation

Block Type	Library
Math Function	Math Operations

QGen Usage Constraints

- "magnitude-2", "conj" and "hermitian" not supported

  The complex-valued functions: "magnitude-2", "conj" and "hermitian" are not supported.

- rem and mod semantics

  The semantics of "rem" and "mod" changed between versions 2011b and 2012a. QGen implements the semantics of 2012a and later. In version 2011b, mod(4,-4) is -4, while in recent versions the result is 0.

Merge

Links to Mathworks Documentation

Block Type	Library
Merge	Signal Routing

QGen Usage Constraints

- Non-empty initial output

  The initial output parameter must not be empty ([]).

- Equal port widths

  The parameter "Allow unequal port widths" must be off.

- Inputs from atomic subsystems

  In case the block inputs come from the same atomic subsystem, the generated Ada code may not be compilable, and the generated C code may have a different behavior than Simulink semantics. One workaround is to merge the signals inside the atomic subsystem instead of outside. Another workaround is to use the `--full-flattening` QGen option to obtain correct Ada and C code generation.

Model Info

Links to Mathworks Documentation

Block Type	Library
Model Info	Model-Wide Utilities

QGen Usage Constraints

- NotGenerated

  QGen supports models using this block to provide a seamless experience in Simulink, however no code is generated for this block. The code generated for the model is not affected.

Model Variants

Links to Mathworks Documentation

Block Type	Library
Model Variants	Ports and Subsystems

QGen Usage Constraints

- Only Overriding Variants are supported

  Model Variant block must have the 'Override variant conditions and use following variant' box checked.

Multiport Switch

Links to Mathworks Documentation

Block Type	Library
Multiport Switch	Signal Routing

QGen Usage Constraints

- Input data dimensions

  Variable-size output signal is not supported, so the dimensions of multiple input data ports should match.

Mux

Links to Mathworks Documentation

Block Type	Library
Mux	Signal Routing

QGen Usage Constraints

- Bus signals cannot be used

  Bus signals are not allowed at the inputs of this block.

  Note: Use BusCreator and BusSelector blocks instead of Mux and Demux to process bus signals, as recommended in the Simulink documentation.

Outport

Links to Mathworks Documentation

Block Type	Library
Outport	Ports and Subsystems

QGen Usage Constraints

- Inheriting initial value from the input signal is not supported

  Parameter 'Source of initial output value' must be set to 'Dialog'.

- Initial output must be specified for conditionally executed subsysems

  If the underspecified initialization mode of the model is 'Simplified' then the 'Initial output parameter' of all outports of conditionally executed subsystems must not be unspecified '[]'.

Product

Links to Mathworks Documentation

Block Type	Library
Product	Math Operations

QGen Usage Constraints

- Matrix inversion is not supported

  This block cannot have one input with the '/' operator if the multiplication type is Matrix(*)

S-Function

Links to Mathworks Documentation

Block Type	Library
S-Function	User-Defined Functions

QGen Usage Constraints

- "Legacy function" supported only with a valid function signature

  If the type (MaskType) of S-Function is "Legacy function", the SFunctionSpec parameter must contain a valid function signature. The function signature is expeted to be in form: [<typename> [<varname>]=]<function> (<arglist>).

- At minimum, a function name or a header file and a function signature must be specified.

  At minimum the information we need is the name of the header file of the external library and a function signature specifying the mapping to block ports and data types of each argument.

- When function signature is not provided, function call has to be added to wrapper module manually

  This option applies when the S-Function block is not generated using the legacy code tool (or one did not provide the full prototype specification). When function prototype specification is not found in the block specification then qgenc will generate a module named <modelname>_wrappers which contains a function for each generated block. An appropriate function call must be added manually in each wrapper function.

Saturation Dynamic

Links to Mathworks Documentation

Block Type	Library
Saturation Dynamic	Discontinuities

QGen Usage Constraints

- No saturation

  The value of the 'Saturate on integer overflow' parameter shall be either empty or 'off'

Scope

Links to Mathworks Documentation

Block Type	Library
Scope	Sinks

QGen Usage Constraints

- NotGenerated

  QGen supports models using this block to provide a seamless experience in Simulink, however no code is generated for this block. The code generated for the model is not affected.

Selector

Links to Mathworks Documentation

Block Type	Library
Selector	Signal Routing

QGen Usage Constraints

- Option "Starting and ending indices (port)" not supported

  Indexing option "Starting and ending indices (port)" is not supported

- Maximum 3 input ports

  The maximum number of input ports is three

- Maximum 2D input data

  Higher than 2-dimensional input data is not supported

- Peculiar combination of parameter options

  With the combination of indexing option "Starting index (port)" and "Output sizes" parameter that has a value over 1, an overflow condition may occur. Simulink does not catch the overflow and QGenc cannot catch it either. An "Index check failed" error is given at run time (in generated Ada).

Shift Arithmetic

Links to Mathworks Documentation

Block Type	Library
Shift Arithmetic	Logic and Bit Operations

QGen Usage Constraints

- Binary Point Shift

  The "Binary points to shift" parameter is ignored by QGen. QGen does not support shifting the binary points for this block as it may generate fixed-point data in output.

- No Floating Point

  Floating-point data types should not be used as input for dynamic bit shift using input port because they may lead to non-integer values which are not accepted by Simulink for this port.

- Supported Size

  For non-scalar values, the size of each dimension of the data of the bit shift number input port or the "Bits to shift: number" parameter should be equal to the size of the corresponding input data dimension.

Signal Specification

Links to Mathworks Documentation

Block Type	Library
Signal Specification	Signal Attributes

QGen Usage Constraints

- Check against 'Dimensions' parameter not supported

  Check against 'Dimensions' parameter not supported: all dimensions should be statically known

Signal Viewer Scope

Links to Mathworks Documentation

Block Type	Library
Signal Viewer Scope	Sinks

QGen Usage Constraints

- NotGenerated

  QGen supports models using this block to provide a seamless experience in Simulink, however no code is generated for this block. The code generated for the model is not affected.

String Constant

Links to Mathworks Documentation

Block Type	Library
String Constant	String

QGen Usage Constraints

- Bounded string types only

  The output data type must be a bounded string, defined by stringtype(x)

Sum

Links to Mathworks Documentation

Block Type	Library
Sum	Math Operations

QGen Usage Constraints

- No sum over dimension mode

  The Sum over specified dimension mode is not supported.

- Matching input/output base types

  Base (element) types of all inputs and output must be the same (follows from MISRA AC SLSF 002 Ver 1.0 2009, Section 3.1.2).

- Accumulator data type constraint

  The accumulator data type must be set to either “Inherit: Same as first input” or “Inherit: Inherit via internal rule” or be the same as the common base type of the inputs and outputs (follows from MISRA AC SLSF 002 Ver 1.0 2009, Section 3.1.2).

Terminator

Links to Mathworks Documentation

Block Type	Library
Terminator	Sinks

QGen Usage Constraints

- NotGenerated

  QGen supports models using this block to provide a seamless experience in Simulink but the block does not affect the code generated for the model. No code is generated for this block.

To Workspace

Links to Mathworks Documentation

Block Type	Library
To Workspace	Sinks

QGen Usage Constraints

- NotGenerated

  QGen supports models using this block to provide a seamless experience in Simulink, however no code is generated for this block. The code generated for the model is not affected.

Transfer Fcn First Order

Links to Mathworks Documentation

Block Type	Library
Transfer Fcn First Order	Discrete

QGen Usage Constraints

- Boolean and integer inputs not supported

  Only floating-point types are supported

Transfer Fcn Lead or Lag

Links to Mathworks Documentation

Block Type	Library
Transfer Fcn Lead or Lag	Discrete

QGen Usage Constraints

- Boolean and integer inputs not supported

  Only floating-point types are supported

Transfer Fcn Real Zero

Links to Mathworks Documentation

Block Type	Library
Transfer Fcn Real Zero	Discrete

QGen Usage Constraints

- Boolean and integer inputs not supported

  Only floating-point types are supported

Trigger

Links to Mathworks Documentation

Block Type	Library
Trigger	Ports and Subsystems

QGen Usage Constraints

- Only function-call triggers are supported in a top-level diagram

  A Trigger block with trigger type other than “function-call” (i.e., “rising”, “falling” or “either”) is not supported in a top-level diagram. If such a data-drive Trigger is needed, then a workaround is to wrap the contents of the top-level diagram into a Triggered Subsystem and pass the model-level signal that is designated for triggering the model’s contents through a data port.

- Initial trigger signal state must be set to “compatibility (no trigger on first evaluation)”

  The property “Initial trigger signal state” must be set to “compatibility (no trigger on first evaluation)”.

- Treat as Simulink function must be “off”

  The property “Treat as Simulink” function must be “off”.

Triggered Subsystem

Links to Mathworks Documentation

Block Type	Library
Triggered Subsystem	Ports and Subsystems

QGen Usage Constraints

- Non-scalar input to trigger port is not supported

  Non-scalar input to trigger port is not supported.

Trigonometric Function

Links to Mathworks Documentation

Block Type	Library
Trigonometric Function	Math Operations

QGen Usage Constraints

- Complex exponential not supported

  Function: “cos + jsin” is not supported (complex-valued operation).

- Approximation method not supported

  The “Approximation method” option must be set to “None”.

Uniform Random Number

Links to Mathworks Documentation

Block Type	Library
Uniform Random Number	Sources

QGen Usage Constraints

- Minimum and Maximum same dimensionality

  Parameter “Minimum” must have the same dimensions as parameter “Maximum”.

UnitConversion

Links to Mathworks Documentation

Block Type	Library
UnitConversion	Signal Attributes

QGen Usage Constraints

- Only a subset of units is supported

  There is a table in <install_dir>/share/qgen/library/supported_units.txt that contains the supported conversions. Users can add new entries to this table. Every entry contains 6 fields: Dimension, Symbol, SI, Scale, Inverse and Offset, where Dimension is the physical variable (Length, Mass, Time, Temperature, Angle, Velocity), Symbol is the string used in Simulink to represent the unit, SI indicates whether it belongs to the unit the International System of Units, Scale is the conversion factor between the unit and its correspondent unit in the International System, Inverse indicates whether we need to divide instead of multiply, and Offset is the offset to add in the conversion.

Wrap To Zero

Links to Mathworks Documentation

Block Type	Library
Wrap To Zero	Discontinuities

QGen Usage Constraints

- WrapToZero Threshold dimensions

  Parameter "Threshold" must have the same same dimensions as input, except if parameter "Threshold" is a scalar.

n-D Lookup Table

Links to Mathworks Documentation

Block Type	Library
n-D Lookup Table	Lookup Tables

QGen Usage Constraints

- Supported interpolation methods

  Flat and Linear interpolation methods are supported. Nearest and Cubic spline methods are not supported.

- Supported extrapolation methods

  Clip and Linear extrapolation methods are supported. Cubic spline method is not supported.

- Maximum number of table dimensions

  Table dimensions higher than 2 are not supported.

8.2.4 Additional Simulink Block Constraints

S-Functions and Legacy Code

See *Integrating external code*.

Lookup and Interpolation Blocks

The blocks where interpolation or lookup table are given by input port (Lookup Table Dynamic, Prelookup,Interpolation Using Prelookup) can not appear in a model where the table data is received from a top-level input. This includes models referenced by Model Reference block.

8.3 Other Simulink Constraints

8.3.1 Model Configuration Parameters

Solver

Solver type should be discrete, fixed-step.

Diagnostics

Model Initialization must be set to "Simplified" or, if it is set to "Classic", then "Detect multiple driving blocks executing at the same time step" must be set to "error".

8.3.2 Blocks Configuration

All blocks should have sample time set to "inherited", except in the case of non-directfeedthrough blocks (UnitDelay, DiscreteTimeIntegrator, ...).

8.3.3 Signal Dimensions

Scalar, vectors and two-dimensional matrices are supported. Matrices with more dimensions are not supported.

8.3.4 Data Dictionaries

Data Dictionaries are supported, however it is mandatory to make sure that each data dictionary of a given model references the data dictionaries of its child referenced models in the section below. This was mandatory until R2018b in Simulink but Simulink R2019a is no longer enforcing this behavior, yet QGen still requires data dictionaries to use the reference mechanism.

8.3.5 Library Factorization

QGen can generate a single function that will be called for all instances of an Library Atomic Subsystem that share the same interface. The Atomic subsystem should also not use any Model Workspace or Data Dictionary data in order to be factorized.

8.3.6 Custom Data Types

It is possible to generate code with externally declared datatypes. See *Custom Data Types* for the details.

8.3.7 Use of Callback Functions

Simulink allows adding callback functions to models, blocks or port. These functions may be there just for documentation purposes. QGenc does not support callbacks functions. It is up to the user to verify they are not present or, if present, they do not affect the model behaviour.

NB! There is an exception in case of masked blocks – if the purpose of a callback function is to transfer mask parameter value from mask to all parameters with same name inside of the masked subsystem, then this is supported by QGen.

8.3.8 Real-valued Computations Only

- QGen supports only integer and floating point -valued signals and computations. Complex values are not supported.
- Fixed-point values are not supported, with an exception when automatically generated by Simulink for the blocks listed below. In such cases, fixed-point data type will be exported to XMI, and later on converted to integer (preserving sign and scaling) in QGen preprocessor.
  - The supported list of blocks is:
  - The supported fixed-point formats are types:
    - starting with the “sfix” or the “ufix” prefix
    - if scaling is used, containing “E” or/and “n” characters in the encoding suffix
    - Refer to Mathworks documentation for more information about fixed-point formatting.
- Parameters that control the signal type, e.g. “Output signal type” must have value “real” or “auto”.

8.3.9 Bus Signals

- QGen supports both virtual and non-virtual buses. For more information about the code generation for buses, see the *Custom Data Types* section.
- Buses are supported at the input-output of following Simulink blocks
  - Bus Creator
  - Bus Selector
  - Bus Assignment
  - Signal Conversion
  - Inport
  - Outport
  - Unit Delay
  - Switch

8.3.10 Forwarding Tables in Libraries

Forwarding Tables can be used in Simulink libraries. There are a few restrictions in their use:

- Transformation Functions are not supported
- The redirection of blocks can only be defined in the library where this block is defined. For example, if we want to map "LibA/BlockA" to "LibB/BlockB", this redirection can only be defined in "LibA".

8.3.11 MISRA Simulink Guidelines

QGen automatically checks that the input model conforms to selected MISRA Simulink constraints. QGenc should not be used as a MISRA checker tool. The checks are limited to constructs which directly influence the code generation policy. For full check one should use a dedicated MISRA checker.

Via the –no-misra option, `qgenc` can cope with models violating the checked MISRA constraints. However, we suggest to adopt at minimum the following rules:

- Blocks that can be configured to use saturation must have this flag off. The Saturation or Saturation Dynamic blocks should be used, if saturation is needed.
- Blocks other than the Rounding block that can be configured to use different rounding options must be set to round towards zero. The only exception is Data Type Conversion, where the rounding option is also supported.
- Data type conversion of block input and output should be done using the Data Type Conversion block and not by other blocks with block specific type conversion options.

8.3.12 Simulink Coder(TM) Parameters

qgenc respects selected parameters from Signal and Parameter objects that do not influence the model semantics and are intended for tuning the Simulink Coder behaviour. The semantics may not be necessarily the same as that assumed by the Simulink coder. Below is the table of supported parameters their values and impact on the generated code.

Table 2: Supported storage classes

Value	Notes	Ada	C
Auto, Default, Global or Simulink-Global	In case of a Signal object, the variable is defined in module corresponding to containing subsystem. A constant corresponding to a Parameter object is defined in <modelname>_params module.	`<name> : [constant] <typename> := <value>;`	`[const] <typename> <name> = <value>;`
Exported-Global	A constant (in case of Parameter) or variable (in case of Signal) is defined in <modelname>_params module.	`<name> : [constant] <typename> := <value>;`	`[const] <typename> <name> = <value>;`
Export-ToFile	The Header file parameter is obligatory. A constant (in case of Parameter) or variable (in case of Signal) is defined in the module defined in Header file parameter.	`<name> : [constant] <typename> := <value>;`	`[const] <typename> <name> = <value>;`
Imported-Extern	A constant (in case of Parameter) or variable (in case of Signal) is defined with "extern" qualifier. If Header file parameter has value, with/include statement is generated for this module	`<name> : [constant] <typename> := <value>; pragma Import (Ada, <name>, "<name>");`	`extern [const] <typename> <name> = <value>;`
Import-From-File	The Header file parameter is obligatory. With/include statement is generated for this module. When generating ada code and the included module has ".h" extension, an additional import declaration is generates in module <modelname>_params	`<name> : [constant] <typename>; pragma Import (C, <name>, "<name>");`	`#include "<Header file>"`
Const	Applies to Parameter object only. Adds "const" qualifier to generated variable. Has effect only in case of –const-as-vars, as by default all parameters are treated as constants.	`<name> : constant <typename> := <value>;`	`const <typename> <name> = <value>;`
Volatile	Adds "volatile" qualifier to generated variable/constant and import statement	`<name> : [constant] <typename> := <value>; pragma Import (Ada, <name>, "<name>"); pragma Volatile (<name>);`	`volatile [const] <typename> <name> = <value>;`
ConstVolatile	Applies to Parameter object only. Adds "const" qualifier to generated variable. Has effect only in case of –const-as-vars, as by default all parameters are treated as constants.	`<name> : constant <typename> := <value>; pragma Import (Ada, <name>, "<name>"); pragma Volatile (<name>);`	`volatile const <typename> <name> = <value>;`
Define	Applies to Parameter object only. Adds "const" qualifier to generated variable in case of Ada. In case of c generates a macro definition	`<name> : constant <typename> := <value>;`	`#define <name> = <value>;`

The Header file parameter is supported for some storage classes as described above. The parameter or signal variable

is defined in the module specified by the headerfile attribute. Depending on the storage class the module is either generated (ExportToFile) or assumed to exist (ImportFromFile, Volatile, Const, ConstVolatile).

Note: We recommend that you use a consistent way to assign storage classes, preferably Signals defined in the workspace of your choice. This provides an explicit definition that is easy to track.

8.4 MATLAB

8.4.1 MATLAB .p and .mat Files

MATLAB .p and .mat files are not supported.

8.4.2 MATLAB Code for Parameters in Workspace Containers and Block Parameters

Dimensions of data

Scalars, vectors and two-dimensional data (matrices) are supported. Data with more dimensions is not supported.

Non-scalar data should not contain casts as part of the contained elements.

Accepted Constructs

QGen will export the data that is used within the set of models targeted by the code generation process, based on the workspace containers specified for each model. To define such data in the base workspace for example, one can use sets of .m files that are manually or automatically loaded as part of the model setup.

Only constant definitions are accepted in .m files. The supported format is:

```
<identifier> = [<data type> () <value> ()];

where

<identifier> := <constant name>[.<structure element name>]
<value> := <literal value> | <expression>
```

Datastructures are defined implicitly by assigning values for each structure element. There is currently no way to attach explicitly defined data type to a constant.

Values can be composed of literal values, references to other constants and expressions including of arithmetic operators and supported functions (see section "MATLAB functions" below).

Annotation delimiter is "%".

Examples

```
% Constant definition, implicit data type
MyIntParam = 15;

% Constant definition, explicit data type
MyDoubleParam = double (100);

% Array with explicit data type
MyArrParam = double ([1 2 3]);

% Definition of a constant structure
MyStructParam.Elem1 = 1;
MyStructParam.Elem2 = [1 2 3];
MyStructParam.Elem3 = MyIntParam;

% Expression using other constants
MyIntParam2 = int8 (MyIntParam + MyDoubleParam);
```

Type Definitions

- Type definitions are not supported in parameter .m-files. However, QGen supports several ways of using custom data types. See *Custom Data Types* for further details.
- The only exception is an implicit type definition of a structure (record) parameter. QGen infers the type definition from the instance definition in the parameter file. See *MATLAB Code for Parameters in Workspace Containers and Block Parameters*.
- Type definitions should be scalars to be accepted by QGen; arrays containing type definitions are rejected.

8.4.3 MATLAB Operators

The following unary operators are supported:

```
+ - ~
```

The following binary operators are supported:

```
+ - * .* / ./ ^ .^ < <= > >= ~= || && :
```

Supported values for ":" are detailed in the *MATLAB Expressions* section below.

8.4.4 MATLAB Functions

The following MATLAB functions are supported in both .m files and Simulink block parameters:

```
abs, acos, asin, atan, atan2, bin2dec, ceil, cos, cosh, exp, floor,
hex2dec, log, log10, min, max, mod, rem, round, sign, sin, sinh, sqrt, tan,
tanh, isinf, isnan, isfinite.
```

Explicit typing/casting expressions are supported for:

```
int8, int16, int32, uint8, uint16, uint32, single, double, logical
```

In addition to numeric literals, the following literals are supported as well:

```
Inf
```

8.4.5 MATLAB Expressions

Expressions containing the colon operator can only be used to get a slice of a vector (without increment value):

```
mySlice = myVector(3:8);
```

The following expressions are **not** supported:

- Matrix partial indexing:

```
myMatrix(:,1);
myMatrix(1:3);
```

- Creation of row vector:

```
MyVector = [10:20];
```

- Vector slice with increment value:

```
mySlice = myVector(1:5:20);
```

In case there is a need for using such subarrays, then you should do the slicing in MATLAB and feed an appropriate subrarray to `qgenc`.

For instance when myMatrix is a two dimensional array and a model contains block parameter with value myMatrix (:,1) (i.e. the parameter is a vector containing the first column from matrix) then:

- the original matrix is

```
myMatrix = [1 2; 3 4];
```

- create a vector variable myVector1 in MATLAB

```
myVector1 = myMatrix(:,1);
```

- copy the vector value to the .m file that is given as an argument to `qgenc`

```
myVector1 = [1 3];
```

- replace the expression "myMatrix(:,1)" in model with "myVector1"
- assign to a structure in array:

```
myVector1(1,2).member = 1;
```

8.4.6 MATLAB cell arrays

Cell arrays in MATLAB workspace are not supported. qgen_export_workspace generates warning and array is not exported. When the array is referenced from Simulink model then code generation fails.

8.4.7 Supported Enumerations

Enumerations should all inherit from a signed Integer superclass to be accepted by QGen.

8.4.8 Char Arrays

Char arrays are not supported when defined in any Workspace container. They can however be used in mask parameters, Simulink.Signals or Parameters properties.

8.4.9 Empty Arrays

In most cases empty arrays will be rejected by QGen, and should not be used to initialize values used in the model.

8.4.10 Evaluable Simulink Expressions

Values evaluated with *slexpr()* are not suppported.

8.5 Stateflow chart modelling rules

Stateflow charts must comply to the following modeling rules and constraints.

8.5.1 Event broadcast for local events shall not to be used

Description: Stateflow event broadcast mechanism for Stateflow local events shall not to be used. Dataflow shall be used for controlling the chart's mode instead.

Note: An exception is broadcasting external events, i.e. events that are received by other parts of the (Simulink) model than the broadcasting chart itself. Such events do not directly affect the behaviour of the chart. The only affect is changing global data. This kind of broadcasting is allowed. Typically the trigger type of the event/port is in such cases "Function call".

Justification: Some forms of event broadcast can lead to non-termination. Broadcasting an event in a transition action is one case that can never lead to non-termination by itself. In other cases static checks could be used to detect potentially looping behaviour.

8.5.2 Output event of a chart shall have the "Function call" trigger type

Description: The trigger type property of an output event of a chart shall have be "Function call".

Justification: Function call events ensure unambiguous and controllable timing of event broadcasts. Output events with edge trigger type can create confusing models, because an event can be triggered several times in Stateflow, but is seen once or not at all in the Simulink part.

Note: Input events having a trigger type "Rising", "Falling" or "Either" are supported by QGen.

8.5.3 [REMOVED]

This constraint was applicable up to version 2.1.2 of QGen. It was removed in version 17 and is no longer relevant in subsequent versions.

This section is preserved for traceability of constraints across different versions of QGen.

8.5.4 [REMOVED]

This constraint was applicable up to version 1.0.1 of QGen. It was removed in version 2.0.1 and is no longer relevant in subsequent versions.

This section is preserved for traceability of constraints across different versions of QGen.

8.5.5 Transition actions shall not be used in graphical functions and pure flow-graph decompositions. Condition actions shall be used instead.

Description: Transition actions shall not be used in graphical functions and pure flow-graph decompositions.

Note: This rule applies to pure flow-graphs only. It **does not** concern OR compositions of states, where transition actions are very appropriate.

Justification: This is a robustness constraint against effectively dead modelling constructs. Transition actions are meaningless in the above-mentioned cases since they are executed only when there is a transition from state to state (some versions of Stateflow allow to specify such actions, but they are never executed). In flow-graph networks condition actions should be used instead.

8.5.6 An OR decomposition must always have an unguarded default transition

Description: An OR decomposition must always have an unguarded default transition. Exception: An OR decomposition with only one substate can exist without a default transition. However, when any default transitions are present, then at least one must be unguarded.

Justification: If this condition is not satisfied then a run-time error due to state inconsistency can result, when a decomposition is entered, since it means that a system is in inconsistent state. When the decomposition consists of one substate only, then there is no need for the default transition. However, default transitions might exist, with different transitions specifying different actions. In the last case one of them must be without a guard to avoid confusion reading the model, as in Stateflow the state is nevertheless entered.

8.5.7 [REMOVED]

This constraint was applicable up to version 17.1 of QGen. It was removed in version 17.2 and is no longer relevant in subsequent versions.

This section is preserved for traceability of constraints across different versions of QGen.

8.5.8 Boxes shall only be used for grouping functions

Description: Boxes shall only be used for grouping functions.

Justification: Justification: Using boxes around states or flow-graph networks creates strange and complex scoping. The logical scoping for executing the chart's sections and the name scopes introduced by boxes do not match. Boxes are fine for grouping graphical functions and truthtable functions.

8.5.9 Only bounded flow-graph loops should be used

Description: Flow-graph loops can cause non-termination of the chart's execution. The loops should be designed with care and preferrably use a simple for-style pattern.

Note: This rule is currently not checked by QGen.

8.5.10 The transition arc shall not leave its logical parent's boundary

Description: The transition arc shall not leave the boundary of the logical parent of the transition (the lowest common ancestor of the source and destination states).

> **Warning:** Stateflow issues a warning about such transition arcs when simulating the model. This rule is **not** checked by QGen. In case such arcs exist the behaviour in Stateflow can have some side-effects that are not taken into account by QGen.

Justification: This is an error-prone pattern. The detailed semantics for such transitions is complicated and unintuitive. Identical or similar behaviour can be obtained by more clear and explicit constructs.

8.5.11 Super step semantics shall not be used

Description: The Stateflow super step semantics is not supported. This option should be turned off in charts supplied to QGen.

8.5.12 Variable-size arrays shall not be used

Description: The variable-size arrays in Stateflow are not supported. This option should be turned off in charts supplied to QGen.

8.5.13 Only C action language shall be used

Description: QGen does not support the MATLAB action language.

8.5.14 Moore charts shall not be used for breaking data loops

Description: It is possible in Simulink/Stateflow to use Moore charts for breaking causality of data feedback loops (algebraic loops). This is currently not supported in QGen.

If a Moore chart is used in a data feedback path make sure there is also some non-direct feedthrough block, such as Unit Delay on the same path.

8.5.15 Simulink functions shall not be used

Description: Using Simulink functions in Stateflow charts is currently not supported by QGen. Similar functionality can be modelled with an external Simulink subsystem that has the necessary data inputs-outputs connected to the chart's io and that is triggered by a function call event from the chart.

8.5.16 History junctions shall not be used

Description: History junctions are not supported by QGen. Their use is also deprecated by the MISRA AC SLSF guide (rule 046) as it complicates interpretation of the model's semantics.

The history can be modelled with an explicit local variable and transitions instead.

8.5.17 [CHANGED] Absolute-time temporal logic shall not be used

Description: The Stateflow absolute-time temporal logic operators and events are currently not supported by QGen.

This constraint has been changed in QGen version 18, where the support for event-based temporal logic was added. In earlier versions neither event-based nor absolute-time temporal logic were supported.

8.5.18 Change detection functions shall not be used

Description: The Stateflow change detection functions: 'hasChanged', 'hasChangedFrom' and 'hasChangedTo' are currently not supported by QGen.

8.5.19 'in <state>' operator shall not be used

Description: The 'in <state>' operator is currently not supported by QGen.

8.5.20 Data stores shall only be accessed through chart's I/O

Description: Defining or accessing data stores directly from Stateflow chart is not supported. If referring or modifying a value of a Data Store is needed in Stateflow, create resp. Data Store Read and Write blocks and connect them to the chart's I/O.

8.5.21 Some C math functions are not supported

Description: Most C math functions are supported. The exceptions are "labs", "atan2", "fmod", "ldexp" and "rand".

8.5.22 Messages and queued communication are not supported

Description: The Stateflow message objects and queued communication are not supported. If queuing is required, then the necessary mechanism should be explicitly modeled.

8.5.23 An AND decomposition must not contain transitions or junctions

Description: Transitions and junctions have proper semantics only for OR decompositions. In some cases Stateflow allows them also in AND decompositions, but this pattern shouldn't be used.

8.5.24 Nested graphical functions are not supported

Description: QGen does not support nested graphical functions.

8.5.25 Boxes are not supported in functions

Description: QGen does not support boxes in graphical functions.

8.5.26 Machine-level data are not supported

Description: QGen does not support defining shared data at the Stateflow root level. Data objects must be defined at the level of a chart or below.

8.5.27 Box-level data are not supported

Description: QGen only supports defining functions in boxes. Local or parameter data may be defined at a function level.

> **Warning:** This rule is currently not checked by QGen. The behaviour of simulation and code generated by QGen can be different.

8.5.28 Simulink states are not supported

Description: QGen does not support the state kind 'Simulink state'.

8.5.29 Custom C code is not supported

Description: QGen ignores any custom code specified in the model and such code cannot be referenced from Stateflow charts.

8.5.30 Literal code (the literal code symbol '$') is not supported

Description: Literal code (the literal code symbol $) is currently not supported in Stateflow actions and guards.

8.5.31 The 'change' ('chg') event is not supported

Description: The implicit 'change' ('chg') event is currently not supported.

8.5.32 The time symbol 't' is not supported

Description: The absolute time symbol 't' is not supported by QGen.

8.5.33 The reference '*' and dereference '&' operators are not supported

Description: These operators (i.e. pointers) are not supported in Stateflow guards and actions.

8.5.34 The shorthand notation for combining multiple state action conditions is not supported

Description: QGen doesn't support the state label notation where multiple action conditions have been combined. E.g., the following are not supported:

```
en, ex: Out1++
du, on e1: Out2++
```

Workaround: Specify each action condition separately and duplicate the action.

```
en: Out1++
ex: Out1++
du: Out2++
on e1: Out2++
```

8.5.35 Explicit type cast operator 'cast' is not supported

Description: QGen doesn't currently support this operator.

8.5.36 The type reference operator 'type' is not supported

Description: QGen doesn't currently support this operator.

8.5.37 [CHANGED] The 'bind' keyword is not supported

Description: QGen does not support the "bind" keyword in state labels.

The constraint is enforced since QGen version 22. In earlier versions the construct was accepted, but ignored by QGen.

8.5.38 The 'Saturate on integer overflow' property of a Stateflow chart must be set to "off"

Description: QGen does not support this feature.

8.5.39 The 'Create output for monitoring' property of a Stateflow chart or state must be set to "off"

Description: QGen does not support this feature.

8.5.40 Stateflow functions with multiple output data definitions are not supported

Description: QGen does not support multiple output data definitions (formal parameters with direction 'out') in Stateflow functions. However, a single non-scalar output is supported. The number of non-scalar input arguments is not limited.

8.5.41 Stateflow functions with non-scalar output must only be used in simple assignments

Description: QGen supports calling Stateflow functions with non-scalar output only from statements of following form:

```
Y = fun(X)
```

Combining and/or nesting such calls with other expressions is currently not supported.

8.5.42 Inlining graphical functions is not supported

Description: QGen currently ignores the 'Function Inline Option' of Stateflow functions and treats them always as if the option was set to "Function".

8.5.43 The first index of non-scalar data must be 0 (default)

Description: The 'First index' property of any Stateflow data object must be set to 0 or be unset (left to default).

8.5.44 Atomic subcharts are not supported

Description: QGen doesn't currently support the option 'Atomic Subchart' in the 'Group & Subchart' properties of state objects. You may, however, use the options 'Group' and 'Subchart'.

Note: This also means that subcharts cannot be referenced from libraries. To reuse charts place the entire chart to a library.

8.5.45 Strong data typing with Simulink I/O must be used

Description: The chart option "Use Strong Data Typing with Simulink I/O" must be set.

8.5.46 Commented out elements are not supported

Description: QGen doesn't support commented out state chart elements, such as states, transitions, functions and junctions.

8.5.47 Object names must not coincide with keywords

Description: The names of user-defined objects must not coincide with any Stateflow keyword. The list of currently known unsupported names is provided in the following section. In some cases using such a name for a user-defined object is supported in Stateflow, but such models have confusing and error-prone semantics.

> **Warning:** This rule is currently not checked by QGen. The behaviour of simulation and code generated by QGen can be different.

Stateflow Keywords

Boolean symbols

- true
- false

Change detection

- hasChanged
- hasChangedFrom

- hasChangedTo

Complex data

- complex
- imag
- real

Data types

- boolean
- double
- int8
- int16
- int32
- single
- uint8
- uint16
- uint32

Data type operations

- cast
- fixdt
- type

Explicit events

- send

Implicit events

- change
- chg
- tick
- wakeup

Messages

- send
- forward
- discard
- isvalid
- length
- receive

Literal symbols

- inf
- t (C charts only)

MATLAB functions and data

- matlab
- ml

State actions

- bind
- du
- during
- en
- entry
- ex
- exit
- on

State activity

- in

Temporal logic

- after
- at
- before
- every
- sec
- msec
- usec
- temporalCount
- elapsed
- t
- duration
- count

C Math Functions

- ‘abs’, ‘fabs’, ‘sin’, ‘cos’, ‘tan’, ‘sinh’, ‘cosh’, ‘tanh’,
- ‘asin’, ‘acos’, ‘atan’, ‘log’, ‘log10’, ‘ceil’, ‘floor’, ‘sqrt’,
- ‘exp’, ‘pow’, ‘fmod’, ‘labs’, ‘ldexp’, ‘atan2’, ‘rand’

8.5.48 Transition paths involving states must not contain flow-graph loops

Description: Transition paths that start or end in a state must not contain a flow-graph loop, i.e. a loop involving only transitions and junctions, on the transition path.

Note: Transition paths that start and end in the same state, but do not involve a flow-graph loop according to the above criteria are not concerned by the current constraint and are allowed.

Flow-graph loops involving only transitions and junctions are, however, allowed in graphical functions and inner compositions of states. Such graphical functions can, for example, also be called from transition or condition actions of transitions having a path to or from a state.

8.5.49 Atomic boxes are not supported

Description: QGen doesn't currently support the option 'Atomic Subchart' in the 'Group & Subchart' properties of box objects. You may, however, use the options 'Group' and 'Subchart'.

Note: This also means that subcharts cannot be referenced from libraries. To reuse charts place the entire chart to a library.

8.5.50 "Data must resolve to signal object" is not supported for local data

Description: QGen doesn't currently support the option "Data must resolve to signal object" for local data. You may, however, use this option for the chart's outputs.

8.5.51 "Treat Exported Functions as Globally Visible" option is not supported

Description: QGen doesn't currently support the chart-level option "Treat Exported Functions as Globally Visible". Functions exported from a chart must be called using a qualified name.

8.6 Target Languages

8.6.1 Ada / SPARK

QGen generates Ada code compliant with the Ada 83, Ada 95, Ada 2005, Ada 2012 and SPARK 2014 standards.

Generated code might be non-compilable when the source of a Merge block is an atomic subsystem. This limitation will be fixed in a future QGen release.

8.6.2 MISRA-C

QGen generates C code compliant with the C89/C99 ANSI standard and MISRA C:2012 guidelines.

8.7 XMI Support

The XMI importer has the following limitations:

- The `xsi:type` of an object is specified using the fully qualified name:

```
<?xml version="1.0" encoding="ASCII"?>
<geneauto.emf.models.gacodemodel:GACodeModel ...>
   <elements xmi:type="geneauto.emf.models.gacodemodel:Module" ... />
<geneauto.emf.models.gacodemodel:GACodeModel>
```

- The `xmi:id` attribute is used to resolve references to elements and is always required for every node. XPath expressions are not (and will not be) supported. For example, the following XMI is acceptable:

```
<expressions
    xmi:type="geneauto.emf.models.gacodemodel.expression:VariableExpression"
    xmi:id="830" ... variable="404"/>
  ...
  <variables
      xmi:type="geneauto.emf.models.common:Variable" xmi:id="404" ... />
```

  Where the `variable` attribute in the first node references the `xmi:id` of the second node.

- If two features are one the eOpposite of the other, the values for both need to be included in the XMI file.

These limitations are not planned to be alleviated.

CHAPTER

NINE

REPORTING SUGGESTIONS AND BUGS

If you would like to make suggestions about QGen or if you encounter a bug, please use GNAT Tracker (from AdaCore's corporate website http://www.adacore.com click on *GNATtracker Access* at the top of the page then enter your username and password), section 'send a report'. Alternatively submit a bug report by email to mailto:qgen@adacore.com, including your customer number #nnn in the subject and a meaningful subject line.

Please try to include a detailed description of the problem, including models to reproduce it if needed, and/or a scenario describing the actions performed to reproduce the problem. If possible, please include the model file and/or the exported XMI files.

The files `$HOME/.qgen/log.*` may also bring some useful information when reporting a bug. The log file is kept under a separate name, `$HOME/.qgen/log.<date_time>`, where <date_time> is the date and time when QGen was executed. Be sure to include the right log file when reporting a bug.

CHAPTER

TEN

APPENDIX A: ERRORS AND WARNINGS

10.1 General notes

This appendix lists the main errors and warnings emitted by the QGen tool in common usage scenarios. There are several different workflows for using the tool. E.g. launching from the Simulink GUI, the `qgen_build` script or explicitly from the system command line. The possible messages vary slightly depending on the chosen workflow.

10.2 Preliminary steps

10.2.1 Exporting model data

The data exporting step is performed via the `qgen_export_xmi` script, which is invoked either from the Simulink GUI, `qgen_build` script or directly from command line.

In order to get typing and scheduling information the script needs to execute the model. Because of that, errors or warnings from Simulink may appear, if the model is not consistent with Simulink constraints.

In addition, the following errors can be detected by the `qgen_export_xmi` script.

Model file not found

```
There is no system named [Model_Name] to open
```

The error occurs, when the indicated model cannot be found. Check that the model name and path to the model were entered correctly and the model exists.

Unsaved changes to the model

```
Model: [Model_Name] is modified please save it before code generation
```

The error occurs, when the given model is open and contains unsaved changes.

Cannot close the model

```
Cannot close the model [Model_Name] because it has been changed.
```

This error can occur, for example, when the simulation executed by the script altered the model (for instance the model contains a callback that modifies block parameters). In this case simply ignore the message and close the model manually.

Model was created with a newer version of Simulink

```
Created with newer version of simulink.
Version in file: <version of loaded model>.
Simulink version: <current version of Simulink>.
```

If the changes between two different Simulink versions are not too big, it might be possible to open a model in an older version of Simulink than the one, where it was created. By default the script rejects such models with a following error message.

In case it is known, that the model works correctly in the current version, then the version check can be switched off. See, the section for producing the XMI files, *Exporting the model information*.

Unable to open output file

```
Unable to open output file "[file_name]"
```

The error occurs, when an invalid filename was specified for output, the output folder does not exist or the file is locked.

Higher than 2-dimensional data

```
Port: "<block name/port type>", Portnumber=<port number>
<Block reference in error message printed as Simulink hyperlink>
DimensionsError:Over2D
Model has <N> -dimensional elements. QGENC supports up to two dimensions.
```

Code generation from data with higher than two dimensions is not supported. Redesign the model to avoid higher-order arrays.

Missing bus definition

```
Bus object:  [bus name] not found from workspace, import the type definition before
starting the export process.
```

If the tool was not able to find the definition for a bus object a following error is generated.

The type definition must be either manually imported to the workspace before running the qgen_export_xmi script or be fed in a .m script given as an argument when exporting the model.

Undefined parameters

```
The model has undefined parameters, run m-file first.
```

Model parameters contain unbound named constants.

Cannot determine fundamental sample time

```
Cannot determine fundamental sample time for the model.
Make sure that the value of the "Stop time" parameter'
allows simulation to run at least two steps.
```

Make sure that the value of the “Stop time” parameter’ allows simulation to run at least two steps.

Callback functions (warning)

```
Custom code detected in the following blocks: [block list]

or

Custom code detected.

The model [Model_Name] contains callback functions. Please make sure that
none of them changes the model behaviour or structure. Run the script with
'-v' to see the list of affected blocks
```

QGenc does not evaluate any of the callback functions in the model. In case any callback functions are detected the qgen_export_xmi script issues a warning. If you are sure the callback do not alter model behaviour then the warning can be safely ignored. Otherwise, the model must be rearranged to achieve the same functionality without manipulating model elements with scripts.

NB! There is an exception in case of masked blocks – if the purpose of a callback function is to transfer mask parameter value from mask to all parameters with same name inside of the masked subsystem, then this is supported by qgenc.

10.3 Code generation

Code generation is performed by the QGen code generator (`qgenc`) tool. The events raised by `qgenc` have following categories:

- INFO – Information to the user about the operations performed by the tool.
- WARNING – Information to the user of a modelling pattern that does not prohibit code generation, but should be reviewed by the user.
- ERROR – Unsupported modelling construct or usage of the tool. All errors result in the tool stopping without generating the output code. It is possible that several ERROR events are caught and processed by the tool before stopping execution.
- FATAL_ERROR – An error of this category stops the tool’s operation immediately.

The next sections describe the main error and warning events generated by `qgenc`.

10.3.1 Command-line arguments

Bad model file

```
[ERROR] Format of input file [Model_File] not recognized.
```

Unsupported target language

```
[ERROR] Unsupported target language [Some_Language]
```

Only Ada and C output is supported.

10.3.2 Model import

Unsupported block type

```
[FATAL ERROR] Could not instantiate block [Block]: Block type [BlockType]
not supported
```

MATLAB Function blocks

```
[WARNING] 'MATLAB Function' not supported. Replacing the block 'Embedded
MATLAB Function' with an empty function call wrapper
```

Blocks containing MATLAB language are not supported. Replace such block in the model by an equivalent subsystem or chart. Or alternatively, provide the implementation of the MATLAB code in the wrapper generated by QGen.

MATLAB functions in block parameters

```
[FATAL ERROR] Use of unsupported MATLAB function [Function]
Block: [Block]
Parameter: [Parameter]
```

This message indicates that a block that is otherwise supported contains a call to a MATLAB function that is not supported. Replace calls to [Function] in the indicated block and parameter with a function that is supported by QGen or use another kind of block.

Over 2-dimensional data

```
[ERROR] Data types with more than 2 dimensions are not supported:
[Element_Ref]
```

Unsupported StorageClass

```
[ERROR] Unsupported StorageClass [StorageClass]: [Ref]
```

Unsupported StorageClass used in the input file and location indicated by Ref.

Parameter or variable with a reserved name

```
[ERROR] [Reserved_Name] : Variable: reserved name cannot be used
```

The indicated variable should be renamed.

Duplicated workspace parameter

```
[ERROR] Invalid lhs for assignment in M-file at [File_Loc].
Simulink object [Name] already exists.
```

Missing initial value in parameter definition

```
[ERROR] Initial value for a parameter must be specified : [File_Loc].
```

Unsupported statement in the provided M-file

```
[ERROR] Non assignment statement in M-file
```

Only definitions in the form of assign statements are supported in M-files.

10.3.3 Pre-processing signal paths

Unconnected signal destination

```
[ERROR] no DstBlock in Line at [Model_Loc]
```

Unconnected signal source

```
[ERROR] unknown input connection [Model_Loc]
```

Broken data flow processing

```
[ERROR] Could not create incoming signals for block [Block]
```

There was an error processing some of the blocks preceeding the indicated block. Review these blocks and errors.

Bad function-call source

```
[ERROR] Out data port [Block]/[Port] does not emit a function call but is
connected to a Trigger Port
```

Implicit data rate conversion

```
[FATAL ERROR] Data flow between two tasks with different frequencies
requires an explicit Rate Transition block
for Element :Signal [Block1] -> [Block2]
```

Data store references across system boundaries

```
[ERROR] MISRA AC SLSF (v1.0) 005.C forbids use of data store to exchange
data across subsystem boundaries: [Data_Store_Block]
```

Note: Violation of this constraint can be treated as error (default), warning or ignored depending on the MISRA options that were selected, when launching the `qgenc` tool.

10.3.4 Pre-processing model/block references

Unresolved library references

```
[FATAL ERROR] Unresolved library reference [Library]/[LibaryBlock]
```

```
[FATAL ERROR] Missing referenced model [Model] Indicate location
with --lib
```

Unsupported block in a referenced model

```
[FATAL ERROR] Could not import block Reference block [Library]/[LibaryBlock]:
it references a block of an unsupported type
     for Element :[Library]/[LibaryBlock]/[UnsupportedBlock]
```

10.3.5 Pre-processing block constraints

Block type specific constraints

```
[ERROR] [Block]: structure check failed.
```

Most block types supported by QGen have block type specific constraints regarding the allowed values for block parameters or context, where the block is used in the model. The constraints are provided in the QGen User Guide. The constraints are checked for each block instance in a pass called `check_structure`. If it fails, an error such as the one presented here is raised. In addition, the structure check usually provides one or more detailed error messages related to the particular constraint that is violated. Such errors are emitted right before the "structure check failed" error.

Using data type conversion outside the Data Type Conversion block

```
[ERROR] MISRA AC SLSF (v1.0) 002 forbids using data type conversion outside
the Data Type Conversion block. Output of the block must have the same base
type as input.
    [Block]
```

This constraint applies for primitive blocks with data feedthrough.

Note: Violation of this constraint can be treated as error (default), warning or ignored depending on the MISRA options that were selected, when launching the `qgenc` tool.

Using saturation outside the Saturation block

```
[ERROR] MISRA AC SLSF (v1.0) 008.A forbids using saturation outside the
Saturation block.
    [Block]
```

Note: Violation of this constraint can be treated as error (default), warning or ignored depending on the MISRA options that were selected, when launching the `qgenc` tool.

Using rounding outside the Rounding block

```
[ERROR] MISRA AC SLSF (v1.0) 008.B requires the rounding behaviour to be
configured to round towards zero in all blocks other than the "Rounding
Function" block.
    [Block]
```

Note: Violation of this constraint can be treated as error (default), warning or ignored depending on the MISRA options that were selected, when launching the `qgenc` tool.

10.3.6 Sequencing

Feedback loop resolution failure

```
[INFO] Cannot generate code due to possible algebraic loop detected among:
[INFO]    Elementary block [Block1]: is direct feedthrough
[INFO]    Elementary block [Block2]: is direct feedthrough
...
[FATAL ERROR] A possible solution is to make a subsystem virtual or to add
a non- directFeedThrough block (like a UnitDelay) between the blocks above.
```

Check and redesign the indicated part of the model.

10.3.7 Stateflow constraints

Containment of events

```
[ERROR] [Model_Ref]: Events can only be contained in a chart. Event
[Event_Name]
```

Local events

```
[ERROR] Unsupported feature: The scope for event [Event_Name] is local.
     for Element :[Event_Name] : Event
```

Referencing events in event conditions

```
[ERROR] Chart input event expected in given context.
    Event: [Event_Name]
    Referred from: [Element_Ref]
```

Only function-call output events

```
[ERROR] Unsupported feature: Trigger type for ouput events can only be
    FUNCTION_CALL_EVENT
        for Element : [Event_Ref]
```

Unresolved output event

```
[ERROR] Unknown broadcast action: [Name]. The current chart does not have
an output event with the given name.
        [Chart_Ref]
        [Action_Ref]
```

Directed (local) broadcasts

```
[ERROR] Unsupported form of broadcast action. A 'send' call is supported
with a single argument only.
        [Action_Ref]
```

Usage of boxes

```
[ERROR] Boxes can only contain boxes, functions or annotations.
```

Data stores in Stateflow

```
[ERROR] Using Data Stores in Stateflow charts is currently not supported.
    [Chart_Ref]
    [Data_Store_Ref]
```

Instead, use Data Store Read or Write blocks connected to the chart's IO.

Unresolved functions

```
[ERROR] Unable to match function: [Function_Name]
    [Chart_Ref]/[Element_Ref]
```

The function with the given name cannot be resolved. Note that Truth table functions, Simulink functions and C math functions are currently not supported by QGen.

Unconnected inputs

```
[ERROR] Unconnected Inport [Port_Name]: cannot generate code
     for Element : [Chart_Ref]
```

Machine level data

```
[ERROR] Machine level data blocks are not supported. Data Block:
[Data_Name]
```

Starting index

```
[ERROR] Variables must use default indexing. First index for [Var_Name] is
[Value].
```

Literal dimension values

```
[ERROR] Dimensions specified by variables are not supported. Variable
[Var_Name] with dimension [Value]
```

Absolute-time temporal logic

```
[ERROR] Absolute time based temporal logic is not yet supported.
    [Expression]
    [Container]
```

'in' operator

```
[ERROR] 'in' operator not supported.
    [Element_Ref]
```

Action syntax

```
[ERROR] [Label_Ref] expected a statement
[ERROR] Error parsing transition label in:
    [Element_Ref]
    [Label]
```

Actions must be formed of valid statements as opposed to an expression such as `a + b`.

Shift operations on signed integers

```
[ERROR] Shift operations are only allowed with unsigned integers:
    [Left, Right Ref]
```

Broadcast in the init phase

```
[ERROR] External event broadcast is generated during the initialization
phase.  This is not allowed.
        [Chart, Event, Action Ref]
```

Referencing inputs from graphical functions during init

```
[ERROR] Chart has 'Execute at init' property set to True and it contains
graphical functions that reference chart's input. This is not supported.
        [Chart, Input, Context Ref]
```

OR decomposition without an unguarded default transition

```
[ERROR] Violation of a modelling rule. An OR decomposition must always have
an unguarded default transition
        [Chart Ref]
```

Initial transitions in AND states

[ERROR] Initial transitions are not allowed for AND states [Element Ref]

Horizontal inter-level transitions

```
[ERROR] Only vertical inter level transitions are allowed
    [Element Ref]
```

Only vertical inter-level transitions (i.e. transitions within a containment hierarchy) are allowed.

Implicit execution order of states and transitions

```
[ERROR] [Chart Ref] does not have userSpecifiedStateTransitionExecutionOrder
    flag set to True
```

Simulink functions

```
[ERROR] Simulink functions are not supported
    [Chart, Function Ref]
```

Transition actions in flow graph compositions

```
[ERROR] Transition actions are not allowed in [Element Ref] : Flow graph
    composition
```

Nested calls to graphical functions returning non-scalar data

```
[FATAL ERROR] Nested calls for non-scalar functions are not supported for
    C code generation: [Function Ref]
```

10.3.8 Code generation

This section describes some generic code generation errors.

Unresolved variable or parameter

```
[ERROR] Can not resolve VariableExpression.
    Expression : [Expression_Ref]
    references variable with name=[NAME], which is not found.
```

The model contains a block parameter or variable that is not found. Make sure all required workspace parameters have been passed to QGen and that the model can be simulated in Simulink.

Enumerated types for Ada code generation

```
[ERROR] Duplicate representation value in an enumeration type not allowed
    for Ada code generation. Please refine the assignment of representation
    values in the type definition and update the model as needed.
        [Enumeration ref, Literal, Other Literal]
```

When the chosen target language is Ada make sure all enumeration literals in one type are associated with different integer values.

Signal resolution to Simulink.Signal objects

```
[WARNING] Not resolving signal '<signal_name>' that matches a defined
    Simulink.Signal because MustResolveToSignalObject is false and
    model signal resolution is 'Explicit only'
for Element :<block_path>
```

When the "Signal resolution" parameter of the model is set to "Explicit only", signal resolution to Simulink.Signal objects will **not** occur for signals that are not explicitly marked for resolution. This warning indicates a case where the user has defined a Simulink Signal with this name as the signal in the model but has not set the "MustResolveToSignalObject" parameter of the signal to true. In this case the variable for the signal will not be resolved to the global Simulink Signal. Typically this is caused when the user has forgot to mark the port for resolution.

The warning can safely be ignored if not marking the signal for resolution was intentional.

Use of mutual recursion

```
[ERROR] Himoco_Wrapper: Unable to sort functions based on the call order in <Module Name>
[ERROR] Mutual recursion detected between functions:
[ERROR] <Function Name>
[ERROR] <Function Name>
```

QGen does not allow the use of mutual recursion in its generated code. Typically this pattern can only be generated if modelled explicitly with Stateflow functions. Mutual recursion is generally considered a bad coding pattern. In this case you should refactor your Stateflow algorithm. Note that direct recursion is allowed.

CHAPTER

ELEVEN

ACKNOWLEDGMENTS

QGen was developed with support from FUI Project P and EuroStars project Hi-MoCo (E6037, EU40149).