Costa Rica Institute of Technology

Czech Technical University in Prague

Undergraduate project

Code generation for automotive rapid prototyping platform using Matlab/Simulink

Author:
Carlos Jenkins

June 24, 2013

Contents

List of Figures

1 Introduction

This document describes the final results of the project "Code generation for automotive rapid prototyping platform using Matlab/Simulink".

1.1 Background

Back in the beginning of 2012 a leading automotive company requested the Czech Technical University to develop a Engine Control Unit (ECU) for automotive applications. Real-Time Systems group at the Department of Control Engineering from the Faculty of Electrical Engineering developed a hardware and Software platform to the needs of this industry. The hardware uses Texas Instruments TMS570LS3137 CPU and is built with automotive standards and interfaces in mind. It uses a real-time operating system and was directly programmed in C.

Nevertheless, in accordance to company policies the Software developed for the engine control unit must be designed in a safe and auditable way. The company has the policy to implement the Software for their system using Model-Based Design:

> Model-Based Design (MBD) is a mathematical and visual method of addressing problems associated with designing complex control, signal processing and communication systems. It is used in many motion control, industrial equipment, aerospace, and automotive applications. Model-based design is a methodology applied in designing embedded software.

In order to meet this requirement an interaction layer between the platform and the Software the company uses, Matlab/Simulink, must be implemented. This document describes the implementation of this interaction system.

1.2 Technologies involved

1. Matlab/Simulink data flow graphical programming language tool for modeling, simulating and analyzing multidomain dynamic systems.
2. Standard ANSI C programming.
3. FreeRTOS real-time operating system.
4. Texas Instruments TI Code Generation Tools (CGT).
5. RPP in-house automotive hardware board using Texas Instruments TMS570LS3137 CPU.

1.3 Objectives

Main objectives of this project are:

1. Allow C code generation from Matlab/Simulink models for custom made hardware platform.
2. Implement model blocks for some of the peripheral units of the board for use in Simulink programming.

At the time of this writing the objectives of this project are considered successfully achieved.

1.4 Benefits

Expected benefits of this project are:

1. Enabling faster implementation and rapid-prototyping of Software components through the use of model-based programming.
2. Enabling better and clearer visualization of Software implementations for the hardware board through models.
3. Improve auditability of Software system for automotive applications.

At the time of this writing the benefits of this project are considered enabled.

1.5 Final outcome

The main products generated for this project are:

- **C Support Library**:
  Define the API to communicate with the board. Include drivers and operating system.
- **Simulink Coder Target**:
  Allows Simulink model's code generation, compilation and download for the board.
- **Simulink Block Library**:
  Set of blocks that allows Simulink models to use board IO and communication peripherals.
- **Simulink Demos Library**:
  Just a bunch of examples of control application in form of Simulink models.

Each of this product is described deeply in the following sections.

1.6 Document layout

The general layout of this document is as follows:

- Project description, objectives and outcome. This section.
- Software and Hardware setup for development, repository layout and programming standards.
- A section for each of the four products delivered, with:
  - Implementation fundamentals.
  - Repository branch description.
  - Product specific aspects.
  - Reference documentation.
- Glossary.
- References.
- Appendices.

2 Project setup

This sections describes the Software and Hardware aspects required to undertake development for this project. It considers:

- Software development environment.
- Hardware reference documentation and wiring for development.
- Repository's general layout.

2.1 Development environment

This section describes the Software environment setup for development.

2.1.1 Operating system

This project was developed on a GNU/Linux operating system. For development it is recommended to use a Debian based operating system for development as most of the tools are easily available from repositories.

Relevant OS information on which this project was developed:

- Ubuntu 12.04.2 LTS AMD64.
- Kernel 3.2.0-48-generic.
- GCC version 4.6.3.

No test for cross-platform interoperability was performed on the code developed. Although care was taken to try to provide platform independent code and tools there is elements known to be Linux dependent. For a list of this elements refer to Appendix B: Known operating-system dependent files.

2.1.2 Version Control System

The version control system used for this project is **git**. The repository of this project contains all the files produced during development, including documentation, references, code and graphics. Also, the GUI application **giggle** was used to easily review changes. To install both execute on a terminal:

```
1 sudo apt-get install git giggle
```

2.1.3 TI Code Composer Studio

Code Composer Studio (CCS) is the official Integrated Development Environment (IDE) for developing applications for Texas Instruments embedded processors. CCS is multiplatform Software based on Eclipse Open Source IDE.

The version used in this project is the 5.3.0. Download and install CCS for Linux from:

http://processors.wiki.ti.com/index.php/Category:Code_Composer_Studio_v5

CCS download requires a valid MyTI account. Tedious. CCS download is about 1.5GB. Once downloaded extract the content of the `tar.gz` archiver and run `css_setup_<version>.bin` script as root. Installation must done as root in order to install driver set.

After installation the application can be executed with:

```
cd <ccs>/ccsv5/eclipse/
./ccstudio
```

If the application fails to start on 64bits systems is because CCS5 is a 32bits application a thus requires 32bits libraries:

```
sudo apt-get install libgtk2.0-0:i386 libxtst6:i386
```

If the application crashes with a segmentation fault edit file:

```
nano <ccs>/ccsv5/eclipse/plugins/com.ti.ccstudio.branding_<version>/
    plugin_customization.ini
```

And change key `org.eclipse.ui/showIntro` to false.

Choose "FREE License - for use with XDS100 JTAG Emulators" on the licensing options. Code download for the board is uses that particular hardware. See Development wiring for more details on this hardware.

CCS include Texas Instruments Code Generation Tools (CGT) (compiler, linker, etc). Simulink code generation requires the CGT to be available in the system, and thus, even if no library development will be done or the IDE is not going to be used CCS is still required.
See `<repo>/rpp/rpp/README.txt` file for more information.

You can find documentation for CGT compiler in `<repo>/ref/armcl.pdf` and for CGT archiver in `<repo>/ref/armar.pdf`.

2.1.4 Matlab/Simulink

Matlab/Simulink version used is R2012b for Linux 64 bits. For in-house development the CVUT should provide a network licensing server descriptor file.

2.1.5 GtkTerm

Most of the interaction with the board for development is done through a RS-232 serial connection. The terminal Software used for communication is called GtkTerm.

The default configuration for the board serial communication module is 9600-8-N-1. Note that the RPP Library test suite is setup to 115200-8-N-1.

To install GtkTerm execute:

```
sudo apt-get install gtkterm
```

2.1.6 Doxygen

Doxygen is the name of the documentation generator used to generate the RPP API documentation based on the source code files. The generated API include dependency graphs and thus it also requires Graphviz, a graph drawing tool. To install both execute:

```
sudo apt-get install doxygen graphviz
```

See API generation on how to use Doxygen to generate the API Reference documentation.

2.1.7 Nested

Nested is the documentation editor used to create the document you're reading. It features a plain text version control friendly simple to read non-cluttered format, WYSIWYM paradigm, divide and conquer document creation approach, a nested (non-linear) document tree and content/presentation separation scheme and thus documents can be published to LaTeX, PDF or HTML. Nested is a tool created by the author of this report.

To install Nested first install dependencies:

```
sudo apt-get install python2.7 python-gtk2 python-webkit python-gtkspellcheck
    texlive-publishers texlive texlive-latex-extra rubber iso-codes subversion
```

Then get the latest revision from the stable repository:

```
svn checkout svn://svn.code.sf.net/p/nestededitor/code/trunk nested
```

Run Nested with:

```
cd nested/nested/
./nested
```

Nested sources for this document can be found on the repository under <repo>/doc/reports/report/.

2.1.8 LMC1

The LMC1 is a simple script developed for this project written in Python 3 using Gtk+ 3.0 Python dynamic bindings PyGObject. This script, based on Michal Horn's command line script, allows to set or clear the outputs of the test board.

This script includes both a GUI and command line tool. If no parameters are given to the script the GUI version is launched:

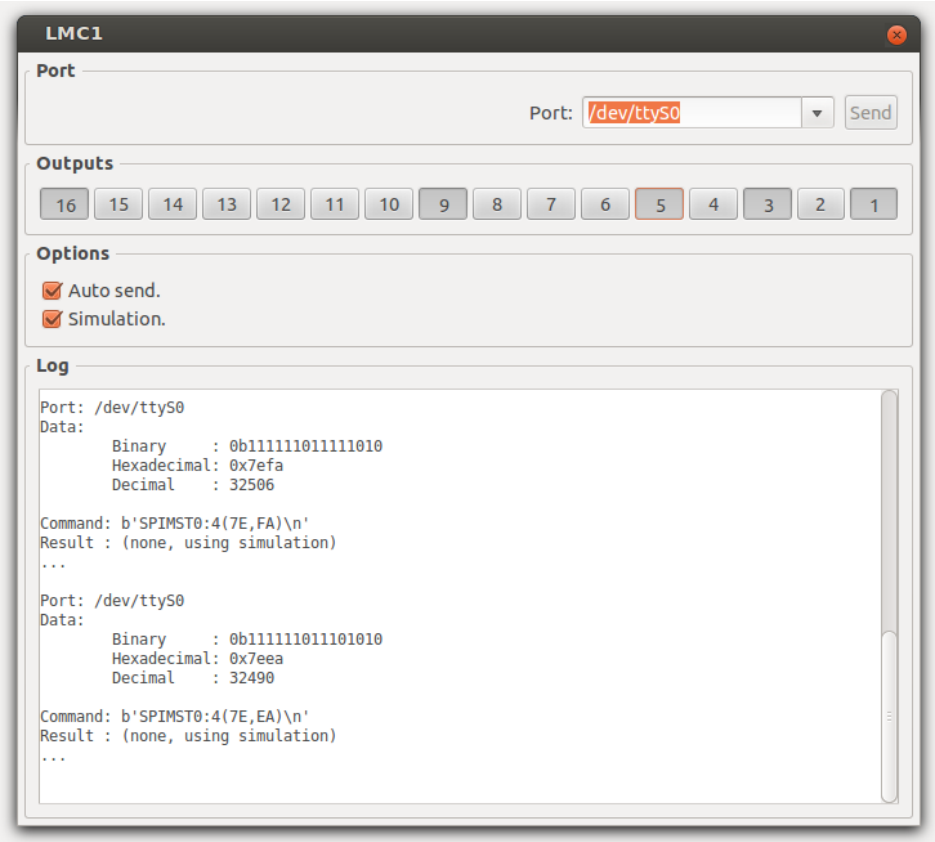


Figure 1: LMC1 GUI application.

To run the LMC1 application first install dependencies:

```
apt-get install python3 python3-gi python3-serial
```

To launch LMC1 GUI version double click file:

`<repo>/rpp/lib/apps/lmc1/lmc1.py`

To launch LMC1 command line version type:

`<repo>/rpp/lib/apps/lmc1/lmc1.py --help`

2.2 Hardware reference

This section provides reference documentation for the RPP board:

- Connectors pinout.
- Modules capabilities and features.
- Wiring configuration for development and testing.

Please note that although this is a hardware reference documentation this is from a Software development perspective and **NOT** Hardware development perspective. For full hardware details please refer to schematics and related documentation.

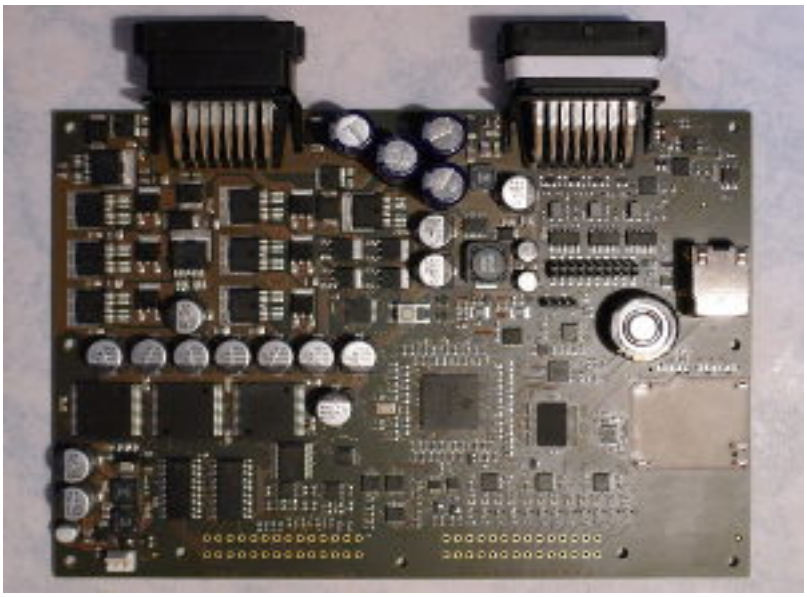


Figure 2: The RPP board (signal connector missing).

2.2.1 Connectors pinout

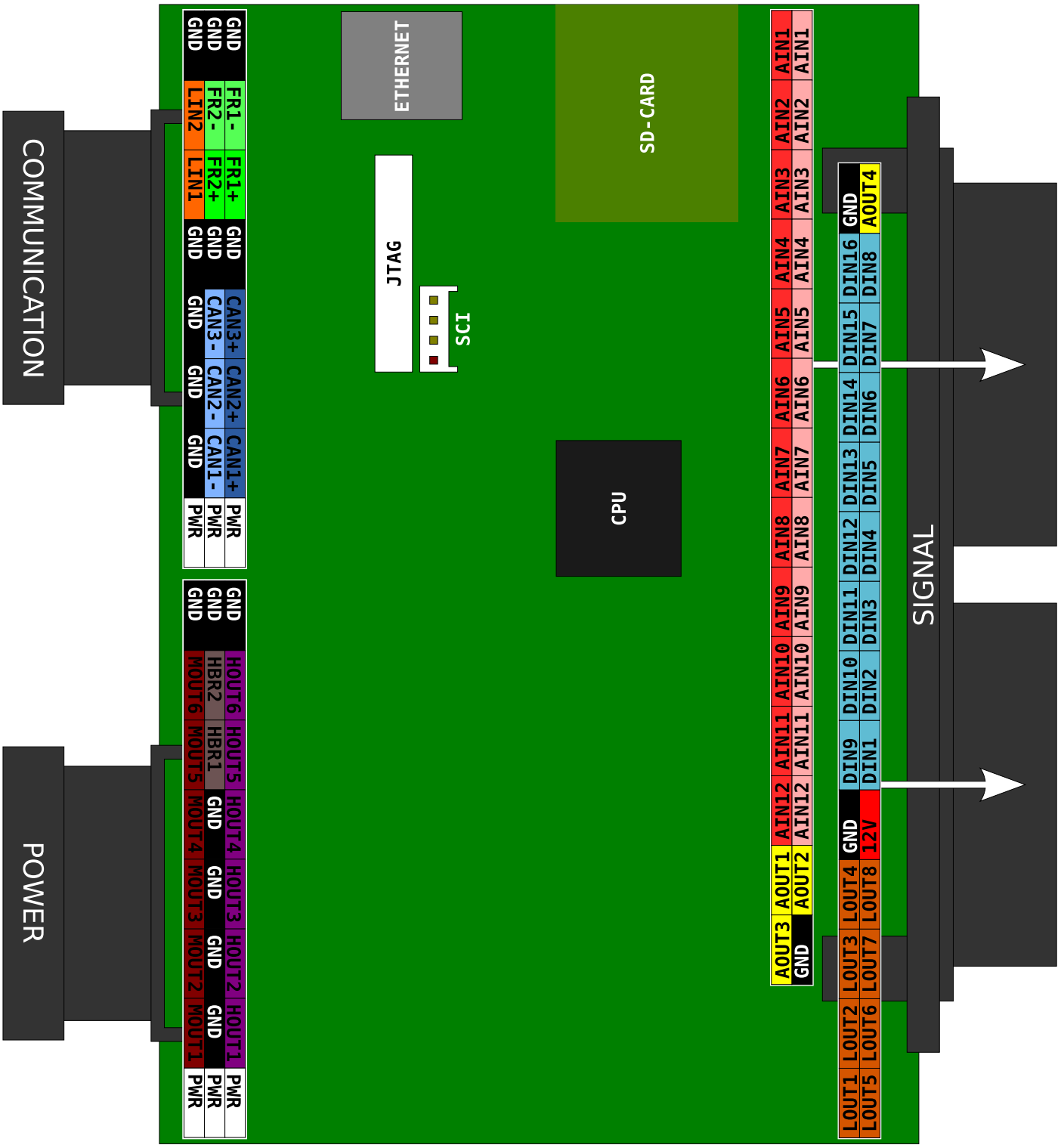


Figure 3: The RPP connectors pinout.

2.2.2 Modules description

This section enumerates the capabilities of the hardware modules from Software perspective.

Logic IO

Digital Inputs (DIN)

- 16 pins available on Signal Connector.
- Pins 9-16 status can be read via GPIO using configurable threshold.
  Pins 9-12 use variable threshold B and pins 13-16 use variable threshold A.
- Variable threshold is a DAC chip MCP4922.
- All pins are read at once via SPI (fixed threshold) using chip MC33972.
- 1-8 are programmable pins and ca be set to pull-up or pull-down. 9-16 are pull-down only.
- All pins can be set to be active or tri-stated.
- All pins can be set to trigger interrupt.
- On-line diagnostic of broken wire.

Digital Outputs (LOUT)

- 8 pins available on Signal Connector.
- Pins for logic output only, up to 100mA.
- All pins are set at once using a chip through SPI.

Analog Input (AIN)

- 12 channels available.
- Differential inputs, thus 24 pins available on Signal Connector.
- Range for 0-20 volts.
- 12 bits resolution.
- Using CPU ADC.

Analog Output (AOUT)

- 4 pins available on Signal Connector.
- Output range is 0-12 volts.
- Using 2 x MCP4922 DACs controlled using SPI.
- Resolution is 12 bits. But because of amplification and voltage reference not all range is used.

Power Output

H-Bridge (HBR)

- 1 port (2 pins) available on Power Connector.
- Communication is done through SPI.
- H-Bridge can be enabled or disabled.
- Current direction can be set.
- PWM control with 1% resolution change of the duty cycle.
- Port can drive load up to 10A.

Power Output (MOUT)

- 6 pins available on Power Connector.
- Pins can drive a load up to 2A. Push/Pull.
- Pins are set using 6 CPU output GPIOs. Diagnostic are read using 6 externally pulled-up open-drain input GPIOs.
- On-line diagnostics. Driver chip will pull-down the corresponding diagnostic pin on the CPU.

High-Power Output (HOUT)

- 6 pins available on Power Connector.
- Pins can be set ON/OFF.
- Pins can drive a load up to 10A with PWM.
- System can read analog values of current flowing (IFBK).
- System can read diagnostics values (DIAG). Detection of a fault condition.

Communication

CAN bus (CAN)

- 3 ports available (CAN uses differential signaling) thus 6 pins are available on Communication connector.
- High speed.
- Recover from error.
- Detection of network errors.

Local Interconnect Network (LIN)

- 2 ports/pins available on Communication Connector.
- Only first port can be used when using the SCI. Second port us shared with SCI.

FlexRay (FR)

- 2 ports available. FlexRay uses differential signaling thus 4 pins are available on Communication Connector.

Serial Comm. Interface (SCI)

- 1 port available inside the box on SCI connector (4 pins).
- Variable baud rate. Tested on 9600 and 115200.
- RS232 standard compatible.

Ethernet (ETH)

- 1 port available. Standard Ethernet connector available inside box.

Data storage/logging

External Memory SD-RAM (SDR)

- 64MB (currently installed) external RAM used for logging. Maximal supported capacity is 256MB.
- Memory test routine available with test Software.

SD Card (SDC)

- Standard SD-Card connector or microSD connector available inside box.
- Communication done using SPI.

2.2.3 Development wiring

For development, the RPP board needs to be wired as follow:

- **Power input**: supply around 13 volts on any PWR pin (Power and Communication connectors) and connect GND to power supply's GND. See Connectors pinout.
- **Serial communication**: board serial interface connected to a RS-232 port on the host computer (`/dev/ttySX`) or to a USB converter (`/dev/ttyUSBX`). **- Debug and code download**: XDS100v2 JTAG Emulator connected to RPP board JTAG connector, which in turn is connected through USB to the host computer (`/dev/ttyUSBX`). See below on details for configuring the XDS100v2 on Linux.

Image of the wiring:

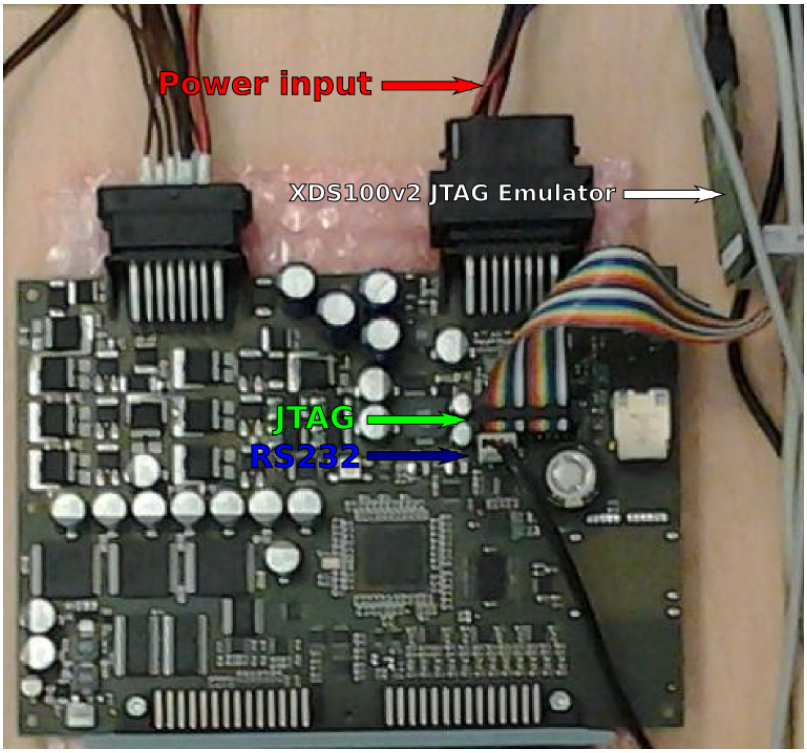


Figure 4: RPP Wiring for Development.

Setup XDS100v2 on Linux:

By default the device (if nothing more connected then `/dev/ttyUSB0`) is added with permissions `664` and `root` as user and group. To access the device write access for current user is required. To do so create a new udev rule file:

```
sudo nano /etc/udev/rules.d/45-pes-rpp.rules
```

And add line:

```
SUBSYSTEM=="usb", ATTR{idVendor}=="0403", ATTR{idProduct}=="a6d0", MODE="0660",
    GROUP="plugdev"
```

Then reload udev rules with:

```
sudo udevadm control --reload-rules
```

To check device properties like `idVendor` or `idProduct` issue the following command:

```
udevadm info -a -p $(udevadm info -q path -n /dev/ttyUSB0)
```

2.2.4 Test wiring

Wiring for test differ from testing objectives. It test performed for this project no communication test, besides SCI, was performed. The following describes how to wire each of the modules tested:

- **DIN**:
  Connect all DIN pins to one LMC1 board outputs.
- **LOUT**:
  Connect all LOUT pins to one LMC1 board inputs.
- **AIN**:
  Connect all low pins to GND and leave all high pins floating. Allow to hook a potentiometer to each pin.
- **AOUT**:
  Connect all 4 pins to different channels on a oscilloscope.
- **HBR**:
  Connect a motor to the H-bridge pins.
- **MOUT**:
  Connect all 6 pins to a LMC1 board inputs. Another option is to connect a motor to one of the outputs.
- **SCI**:
  Connect the SCI to a host computer. See Development wiring.
- **SDR**:
  No particular wiring is required for testing the SD-RAM.

It is recommended to setup a power bus using regulated 12volts from Signal Connector. Power the LMC1 boards and the potentiometer with this bus. The LCM1 controller board should be connected to the host computer through a RS-232 port or a USB converter.

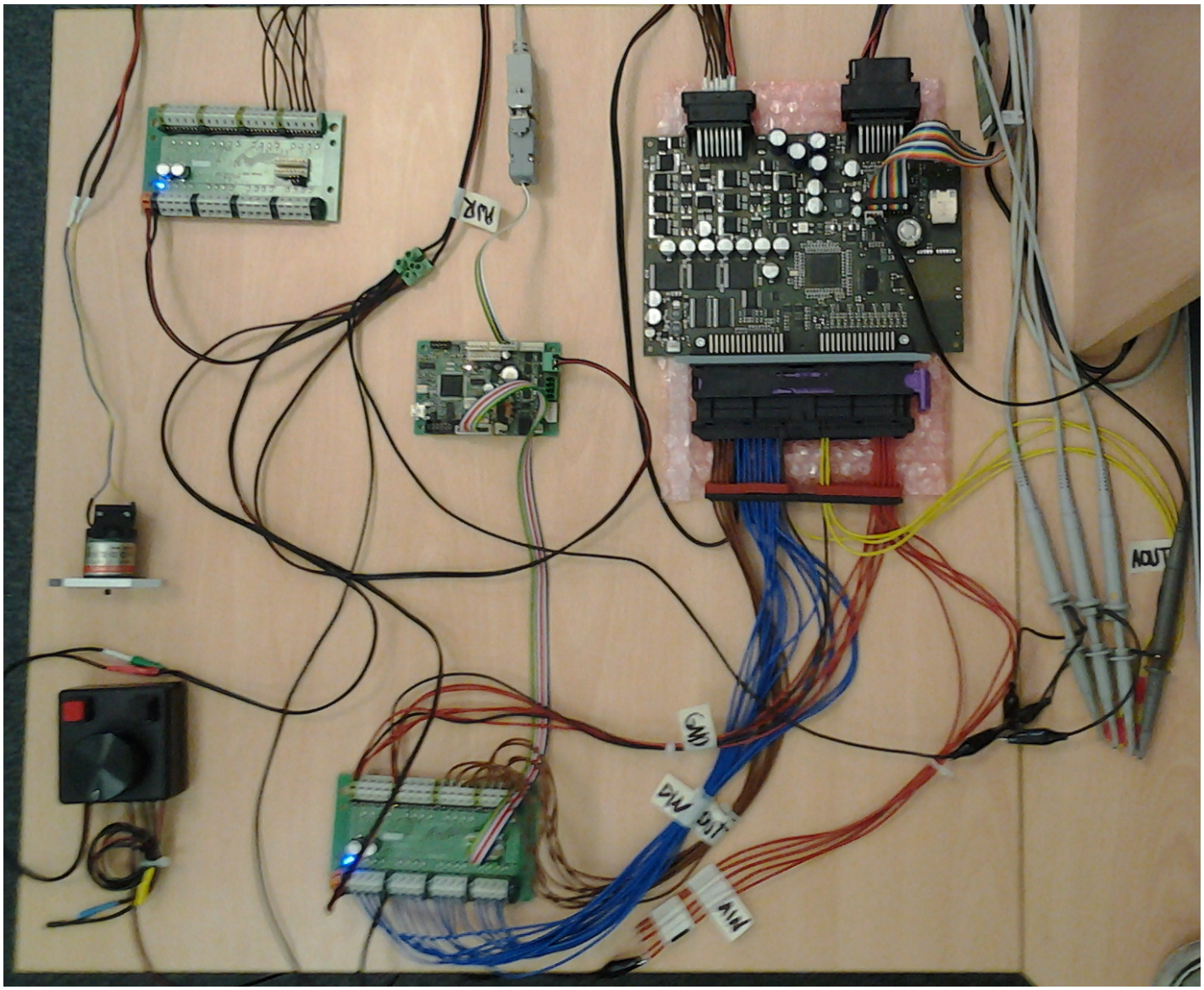


Figure 5: RPP Wiring for Testing.

2.3 Project repository

This git repository holds all the work done on this project.

To get the repository:

```
git clone ssh://git@rtime.felk.cvut.cz/jenkicar/rpp-simulink.git
```

This is a private repository, you require your SSH private key to be authorized. For access please consult the Real-Time Systems Group, Department of Control Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague. For details about this git server refer to:

http://rtime.felk.cvut.cz/hw/index.php/Git_repository_on_this_server

The general layout of the repository is:

```
.
|__ doc         - Documentation created for this project.
|__ refs        - Official reference documentation.
\__ rpp
    |__ blocks  - Simulink Block Set.
    |__ demos   - Simulink Demos Library.
    |__ lib     - C support Library and API.
    \__ rpp     - Simulink Coder Target.
```

A detailed description of the content of each subfolder under `rpp/` can be found in the section *Subdirectory content description* on each dedicated section for the products developed.

In this document, the root folder on this repository is used as reference for file location and is referred with the token <`repo`>.

3 C Support Library

The RPP C Support Library define the API to communicate with the board. It include drivers and operating system. This section documents the implementation of this library.

3.1 Description

The RPP Library is the support library used by Simulink models. It is designed from the board user perspective and exposes a simplified high-level API to handle the board's peripheral modules in a safe manner.

The library as a concept and as a functional unit was introduced by this project. At the beginning of this project the RPP board had just one application developed for. This application intended for board testing allows the user to issue low-level commands to control and test the peripherals of the board. This application was created using a combination of custom code, contributed drivers and generated code from TI tool HalCoGen. Library functionality, like drivers and hardware access, and application logic, like command processor and test routines, was largely merged in a single layer, 166 source code files long highly coupled application. In order to develop independent applications for the RPP board, as it was expected to be each Simulink model, the library logic needed to be separated from the application logic. This work implied a heavy refactoring on the testing application in order extract from it the library functionality. Because the application files were highly coupled in a single layer the refactoring and testing of the library implied roughly 70% of the work done on this project.

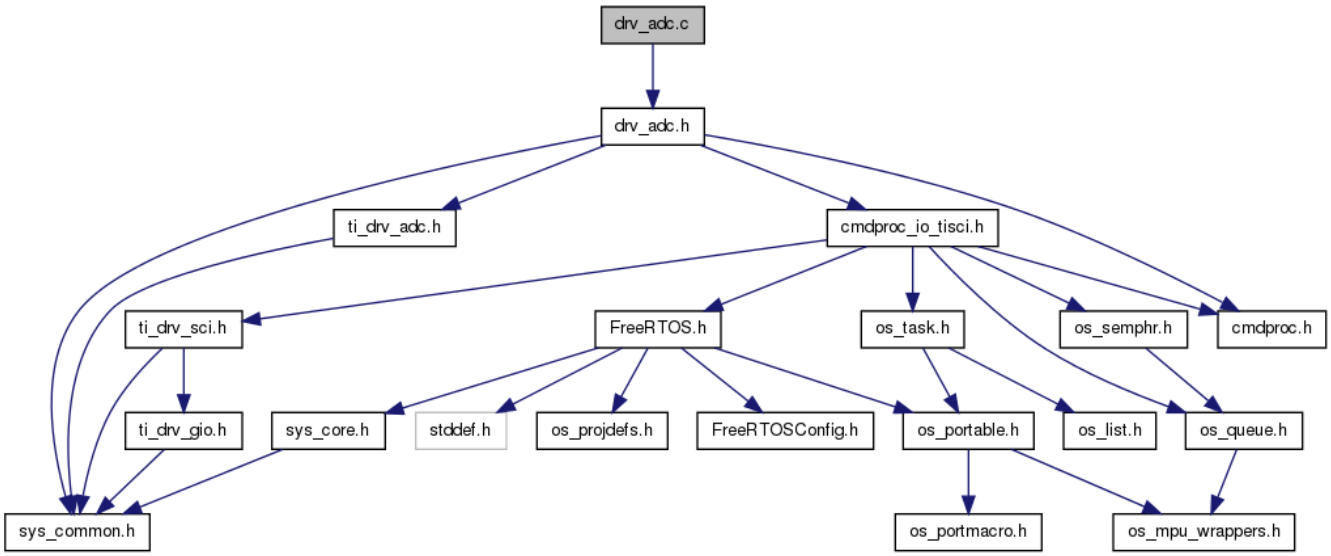


Figure 6: Dependency graph of the ADC driver before refactoring.

The above graph shows the dependencies of the ADC driver before the refactoring. Please note the dependency on `cmdproc_io_tisci.h` and `cmdproc.h`, both application level modules. Also, note the indirect dependency on the Operating System is being resolved through the application modules.

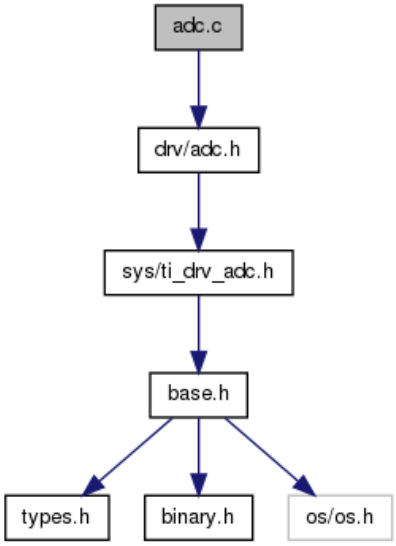


Figure 7: Dependency graph of the ADC driver after refactoring.

The above graph shows the current dependencies for the ADC driver in the RPP Library. Please note that it dependents only on the system layer low-level driver and that the Operating System indirect dependency is resolved through the library foundations `base.h`.

Some other relevant changes introduced with the refactoring are:

- ADC driver was completely rewritten.
- MOUT driver was implemented.
- DIN driver was slightly modified and extended.
- DAC driver was slightly modified.
- HBR driver was largely modified (in particular watchdog functionality).
- SCI driver was refactored and extended.
- SDR driver was implemented.

Also, once the library functionality could be isolated, the resulting API was too low-level to be used by applications, in consequence one of the contributions of this projects was the implementation of a high-level API on top of this low level API: the RPP Layer.

3.1.1 Architecture

The RPP library was structured into 5 layers with the following guidelines:

- Top-down dependency only. No lower layer depends on anything from upper layers.
- 1-1 layer dependency only. The top layer depends exclusively on the bottom layer, not on any lower level layer (except for a couple of exceptions).
- Each layer should provide a unified layer interface (`rpp.h`, `drv.h`, `hal.h`, `sys.h` and `os.h`), so top layers depends on that layer interface and not on individual elements from that layer.

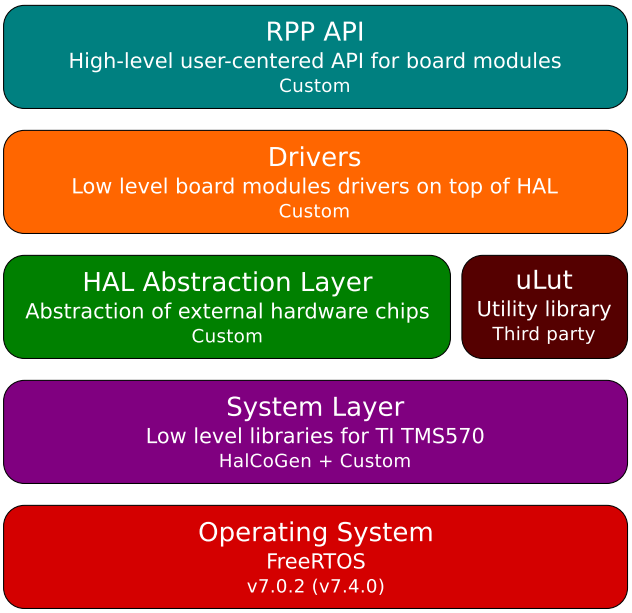


Figure 8: The RPP library layers.

As a consequence of this division the source code files and interface files are now placed on private directories so the previous prefix based inclusion `drv_din.h` is replaced by `drv/din.h`. With this organization user applications only needs to include the top layer interface file (`rpp/rpp.h`) to be able to use the library API.

Please note the sublayer uLut, which is used only by the SPI driver in order to use thread safe queue mechanisms. Because the FreeRTOS already provides thread safe queues and in order to match the order parts of the system it would be advisable to drop this dependency in the future.

3.1.2 RPP Layer Modules

The RPP Layer was structured into 14 different modules from 4 different categories that match the hardware modules on the board:

Category	**Description**	**MNEMONIC**
Logic IO	Digital Input	`[DIN ]`
	Digital (Logic) Output	`[LOUT]`
	Analog Input	`[AIN ]`
	Analog Output	`[AOUT]`
Power output	H-Bridge output	`[HBR ]`
	Power output (12V, 2A)	`[MOUT]`
	High-Power output (12V, 10A)	`[HOUT]`
Communication	CAN Bus	`[CAN ]`
	LIN (Local Interconnect Network)	`[LIN ]`
	FlexRay	`[FR ]`
	Serial Communication Interface	`[SCI ]`
	Ethernet	`[ETH ]`
Logging	SD Card	`[SDC ]`
	SD-RAM	`[SDR ]`

Please note the mnemonic of each module, as they are constantly used on the Software and documentation. Also note that only the following modules where implemented as part of this project:

- DIN.
- LOUT.
- AIN.
- AOUT.
- HBR.
- MOUT.
- SCI.
- SDR.

Modules for which there is a low-level API available on the library but no high-level module was implemented:

- CAN.
- LIN.
- FR.

Modules that are not yet available on the library at all:

- ETH (in the works).
- SDC.
- HOUT (partial).

The following graphic shows the library modules and the connectors on the hardware they map to.

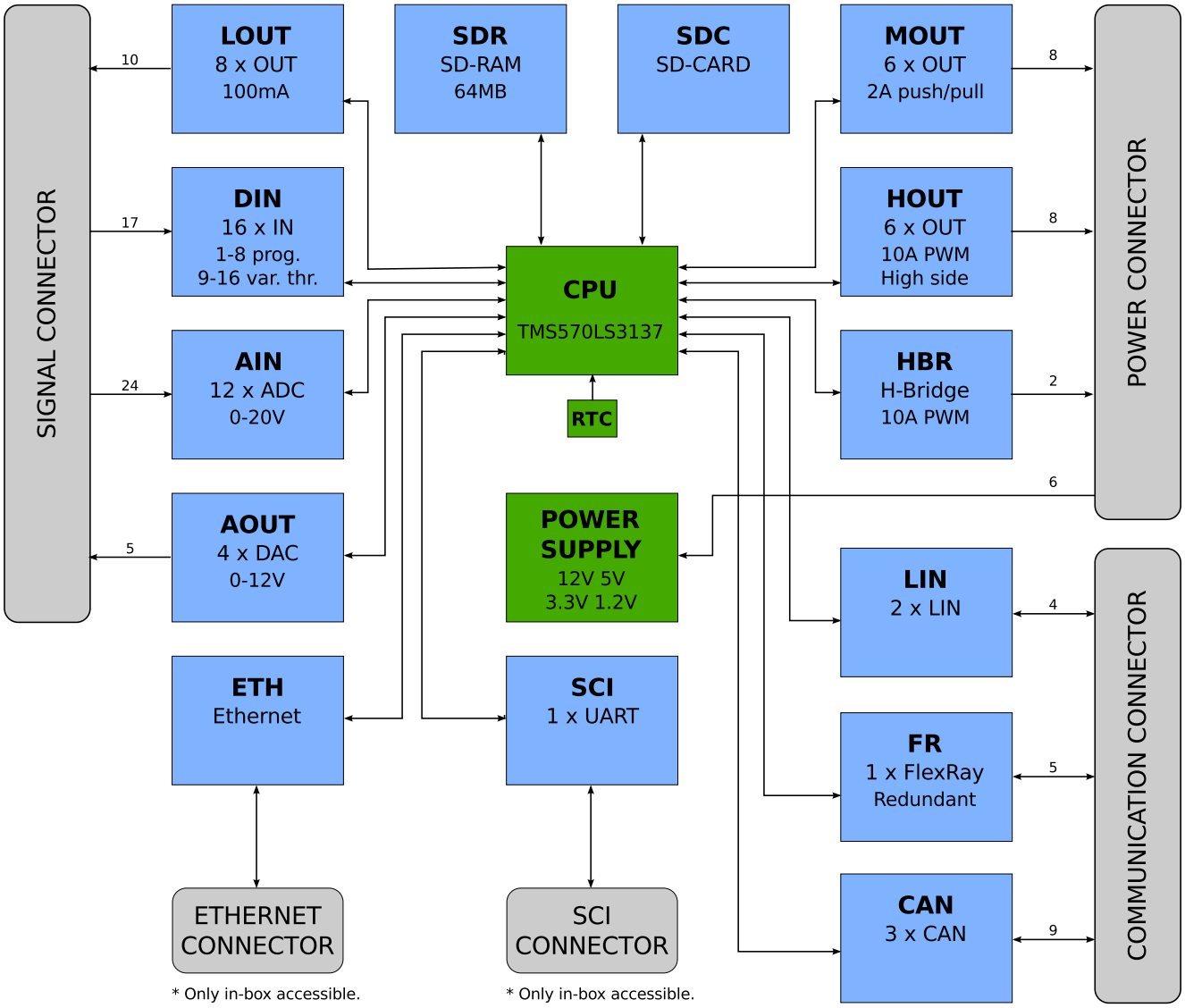


Figure 9: The RPP Library modules.

3.1.3 OS interchangeable layer

The OS Layer is composed by the FreeRTOS source code files. Because the FreeRTOS exposes an stable API the OS layer can be changed in order to upgrade the Operating System or use a different port of the OS, without changing the upper layers source code. The OS Layers currently available for the RPP Library at <repo>/rpp/lib/os/ at the time of this writing are:

- Version 6.0.4 using POSIX port. This layer is the one that should be used when compiling a program for x86(_64) simulation. The port uses the `pthread` library and because of this the port is not true real time and this is considered a simulator.
- Version 7.0.2 using HalCoGen port for TMS570. This layer is the one currently supported and tested. It was originally included in the testing application and was generated by an older version of TI code generation tool HalCoGen.
- Version 7.4.0 using HalCoGen port for TMS570. This layer was extracted from a newly generated project using a newer version of HalCoGen. This layer is untested but *should* work out of the box.
- Version 7.4.2 using ARM Cortex R4 official port for CCS. This layer was created from vanilla FreeRTOS 7.4.2 release. It is tested but non-working. Ticks are proved to be executed in time but applications using this kernel runs at full-speed. The reason if this is currently unknown.

The general layout of all the layers are as following:

- Common source code (kernel):

  ```
  src/os/croutine.c (Optional)
  src/os/list.c
  src/os/queue.c
  src/os/tasks.c
  src/os/timers.c (Optional)
  ```

  Originally found in vanilla distribution in: <FreeRTOSRoot>/FreeRTOS/Source

- Common interface files:

  ```
  include/os/croutine.h
  include/os/FreeRTOS.h
  include/os/list.h
  include/os/mpu_wrappers.h
  include/os/portable.h (with minor editions)
  include/os/projdefs.h
  include/os/queue.h
  include/os/semphr.h
  include/os/StackMacros.h
  include/os/task.h
  include/os/timers.h
  ```

  Originally found in vanilla distribution in: <FreeRTOSRoot>/FreeRTOS/Source/include

- Memory management file:

  ```
  src/os/heap.c (One of 4 version available, see Appendix A).
  ```

  Originally found in vanilla distribution in: <FreeRTOSRoot>/FreeRTOS/Source/portable/MemMang

- Port specific files:

  ```
  src/os/port.c
  src/os/portASM.asm
  include/os/portmacro.h
  include/os/FreeRTOSConfig.h
  ```

  This depend of the port. In the case of the 7.4.2 TMS570 / ARM Cortex R4 for CCS port:

  - First three files can be found in vanilla distribution in <FreeRTOSRoot>/FreeRTOS/Source/portable/CCS/ARM_Cortex-R4.
  - Last file in <FreeRTOSRoot>/FreeRTOS/Demo/CORTEX_R4_RM48_TMS570_CCS5.

In general, the following changes were applied to the source code base of all kernels:

- Replaced include directives to adapt to RPP library standard:
  `#include "` with `#include "os/`
- Line ending character set to UNIX '\n' and tabs replaced by 4 spaces.

3.1.4 API development guidelines

The following are the development guidelines use for developing the RPP API:

- User documentation should be placed in header files, not in source code, and should be Doxygen formatted using autobrief. Documentation for each function present is mandatory.
- Function declarations on the headers files is for public functions only. Do not declare local/static/private functions on the header.
- Documentation on source code files should be non-doxygen formatted and intended for developers, not users. Documentation here is optional and at the discretion of the developer.
- Always use standard data types for IO when possible. Use custom structs as very last resort.
- Use prefix based functions names to avoid clash. The prefix is of the form `[layer]_[module]_`, for example `rpp_din_update()` for the update function of the DIN module in the RPP Layer.
- To be very careful about symbol export. Because it is used as a static library the modules should not export any symbol that is not intended to be used (function) or `extern`'ed (variable) from application. As a rule of thumb declare all global variables as static.
- Only the RPP Layer symbols are available to user applications. All information related to lower layers is hidden for the application. This is accomplished by conditionally including the layers elements on the implementations files only and never on the interface files. Never expose any other layer to the application or the the whole system below that layer will be exposed. In other words, never `#include "foo/foo.h"` in any RPP Layer interface file.
- Any module is conditionally included by using `rppCONFIG_INCLUDE_{MNEMONIC}` directive on the `RppConfig.h` configuration file.

3.1.5 Further improvements

The following are recommendations for future improvements of the library:

- General code revision to remove local-only methods and variables from being exported.
- General code revision and refactoring to normalize the functions naming scheme. Normalize DRV and HAL to use prefix based scheme, not all the functions and exported variables do. Refactor the SYS layer, most of it generated by HalCoGen and that uses `thisNamingScheme` to use library standards (see RPP API programming standards).
- Simplify doxygen documentation on the SYS layer, because is clunky, doesn't add any value and is repetitive. Move it the header files.
- Remove error throwing from wrong parameter input in the DRV layer and assume a *correct parameter and continue* safe approach. Move all error throwing and validation to the RPP layer (already implemented).

Recommendations for changes on the electrical diagrams:

- Change name of GPIO MOUT1_EN to MOUT1_DIAG.
- Change name of GPIO MOUT1_IN to MOUT1_EN.

The current names are misleading.

3.2 Subdirectory content description

→ `librpp.a` and `rpp-lib.lib`

Version controlled RPP static libraries.

The first one is for POSIX simulation, the second one for Simulink models and other ARM/TMS570 applications. This files are placed here by the projects `apps/rpp-lib_posix` and `apps/rpp-lib` when built.

→ `apps/`

Applications related to the RPP library.

This include the CCS studio project for generation of the static library and the test suite. See Static libraries, Test Suite and Base application for more information.

→ `os/`

OS layers directory.

See OS interchangeable layer for more information.

→ `rpp/`

Main directory for the RPP Library.

→ `rpp/doc/`

Documentation directory for the RPP Library. See API generation for more information.

→ `rpp/TMS570LS3137.ccxml`

Descriptor for code download.

This file is used by all the projects including the Simulink RPP Target for code download. It is configured to use the Texas Instruments XDS100v2 USB Emulator. See Development wiring for information about this hardware.

→ `rpp/TMS570LS313xFlashLnk.cmd`

CGT Linker command file.

This file is used by all applications linking for the board, including the Simulink models, static library and test suite. It includes instructions for the CGT Linker on where to place sections and size of some sections.

→ `rpp/include/{layer}` and `rpp/src/{layer}`

Interface files and implementations files for given `{layer}`. See below for details on the RPP Layer.

→ `rpp/include/rpp/rpp.h`

Main library header file.

To use this library just include this file and this file only. Also, before using any library function please call `rpp_init()` function for hardware initialization.

→ `rpp/include/rpp/RppConfig.h`

Library configuration file.

Please refer to the API documentation and header file comments for specific documentation for each configuration parameter.

→ `rpp/include/rpp/rpp_{mnemonic}.h`

Header file for {`mnemonic`} module.

This files includes function definitions, pin definitions, etc, specific to {mnemonic} module. The inclusion of this header can be configured in `RppConfig.h` using `rppCONFIG_INCLUDE_{MNEMONIC}` directive. See API development guidelines.

→ `rpp/src/rpp/rpp_{mnemonic}.c`

Module implementation.

Implementation of `rpp_{mnemonic}.h`'s functions on top of the DRV library. See API development guidelines.

→ `rpp/src/rpp/rpp.c`

Implementation of library-wide functions.

3.3 Test Suite

The `rpp-test-suite` is a RPP application developed as part of this project that includes a series of test tasks or test commands to verify the correct behavior and functionality of the RPP layer modules. There is one command per module, and the command use the same mnemonic that the module.

This test suite can be found in `<repo>/rpp/lib/apps/rpp-test-suite` for the ARM version and in `<repo>/rpp/lib/apps/rpp-test-suite_posix` for the simulated version.

The application enables a command processor using the SCI at **115200-8-N-1**:

```
RPP Library Test Suite.
============================================================
[Type a module to test or 'help']
--> help
Available commands:
        help - Display this help.
        ain  - Test Analog Input.
        aout - Test Analog Output.
        can  - Test CAN communication.
        din  - Test Digital Inputs.
        eth  - Test Ethernet communication.
        fr   - Test FlexRay communication.
        hbr  - Test H-Bridge.
        hout - Test High Power Output.
        lin  - Test LIN communication.
        lout - Test Digital Outputs.
        mout - Test Power Outputs.
        sci  - Test Serial Communication Interface.
        sdc  - Test SD-Card.
        sdr  - Test SD-RAM.
```

Current modules with tests implemented are:

- AIN.
- AOUT.
- DIN.
- HBR.
- LOUT.
- MOUT.
- SCI. (the test-suite itself)
- SDR.

A note of warning: tests spawn OS tasks at the beginning of the test and deletes them at the end. Because current memory memory management implementation cannot free memory the test suite will fill all the memory and tests will be unable to start. In this case just reset the board. See Appendix A: Notes on FreeRTOS memory management for more information.

3.3.1 AIN test description

This test will read all the analog inputs at a rate of 100 times per second and print the result.

```
--> ain
Analog Inputs Test [1-12]:
============================================================
   1    2    3    4    5    6    7    8    9   10   11   12
   0    0    0    0    0    0    0    0    0    0    0    0
```

Status: **PASSED** for channels 1-5. 6-12 remain untested but they *should* work.

3.3.2 AOUT test description

This test will generate a 10Hz sinus wave on all the analog outputs with a sampling rate of 1kHz. The sinus wave of each analog output channel is sifted by (1/4)pi.

```
--> aout
Analog Output Test at 10 Hz:
============================================================
Samples: 7331
```

Status: **PASSED**.

3.3.3 DIN test description

This test will read all 16 + 8 digital inputs at a rate of 100 times per second, using both low speed SPI chip and variable threshold high-speed inputs.

```
--> din
Digital Inputs Test [1-16]:
============================================================
 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16  A  B  C  D  E  F  G  H
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
```

Status:

- Low speed fixed threshold [1-16]: **PASSED**.
- High speed variable threshold [A-H]: **PASSED**.

3.3.4 HBR test description

This test will generate a sinus wave to control the H-Bridge of one period per 20 seconds (0.05Hz) at a sampling rate of 20Hz.

```
--> hbr
H-Bridge Test at 0.05 Hz:
============================================================
Samples: 72
```

Status: **PASSED**.

3.3.5 LOUT test description

This test will show in the digital outputs the value in binary of a counter, incrementing the counter once per second. The counter is 8 bits, the same as the outputs, so 255 seconds are required for an overflow/restart of the counting.

```
--> lout
Digital Output Test:
============================================================
Counter:   40
```

Status: **PASSED**.

3.3.6 MOUT test description

This test will toggle the power outputs one by one per second, then wait 10 seconds in that state while constantly verifying the diagnostics.

```
--> mout
Power Output Test:
============================================================
1     2     3     4     5     6
1: OK 1: OK 1: OK 1: OK 1: OK 1: OK
```

Status: **PASSED**.

3.3.7 SCI test description

A more comprehensive test is not implemented. The very use of this test-suite implies the correct function of the SCI module. Nevertheless, as a future improvement, a test that will verify run-time baud rate changes and test some other RPP SCI functions is desirable.

```
--> sci
You're using the SCI, reading this and typing this command.
Press any key to continue...
```

Status: **PASSED**.

3.3.8 SDR test description

This test will launch a noise generator task that will log noise and then start the library included SD-RAM logging command processor, allowing the user to see and handle the log on the SD-RAM.

```
--> sdr
Log control: 1024kB available.
============================================================
--> log
[   1239864] This is the noise generator at iteration 1 putting some noise value 279735017.
[   1240779] This is the noise generator at iteration 2 putting some noise value 1943579783.

--> available
1023 kB of 1024 kB available.

--> clear
Done.

--> exit
```

Status: **PASSED**.

3.4 Static libraries

The RPP Library can be compiled as a static library for ARM using TI CGT and for x86(_64) using GCC. CCS projects `rpp-lib` and `rpp-lib_posix` in <repo>/rpp/lib/apps/ allows to generate the static libraries. After compilation, as part of the build process, both projects will automatically update the version-controlled static libraries in <repo>/rpp/lib/:

- `rpp-lib.lib`, static library for ARM using TI naming scheme.
- `librpp.a`, static library for x86(_64) using standard Linux naming scheme.

One future improvement would be the creation of a Makefile for each compilation scheme in order to not depend on CCS managed build system. For ARM manual compilation or makefile creation using Texas CGT see the `target_tools.mk` file under the Simulink RPP Target folder. The relevant aspects for compiling and linking an application using the static libraries are:

ARM compilation using CCS for the RPP board:

- Include headers files of the OS for which the library was compiled against. At the time of this writing the OS is FreeRTOS 7.0.2. See OS interchangeable layer section above.
- Include header files for the RPP library.
- Add library `rpp-lib.lib` to the linker libraries. The RPP library **MUST** be looked for before Texas Instruments support library `rtsv7R4_T_be_v3D16_eabi.lib`.
- Configure linker to retain `.intvecs` section from RPP Library:
  `--retain="rpp-lib.lib<sys_intvecs.obj>(.intvecs)"`
- Use the provided linker command file `TMS570LS313xFlashLnk.cmd`.

x86(_64) compilation using GCC for Simulation:

- Include headers files of the OS for Simulation. At the time of this writing the OS is POSIX FreeRTOS 6.0.4.
- Include header files for the RPP library.
- Create a `RppConfig.h` override file and drop DRV layer dependency: `rppCONFIG_DRV 0`.
- Includes must be configured in a way that the `RppConfig.h` taken under consideration is the override and not the library one.
- Add library `librpp.a` to the linker libraries.
- Add `pthread` to the linker libraries.

As an important note, all the models compiled using Simulink will link against `rpp-lib.lib`. When compiling a Simulink model, Simulink, and then `make`, will not update the generated binary if the model hasn't changed, and then if the source code hasn't changed. Static libraries changes are not considered for re-compilation and re-linking. If library development is being done and static library is updated, in order for the Simulink model to generate a newly linked version of the binary the whole code generation folder needs to be deleted in order to force code generation, compilation and linking with the new static library.

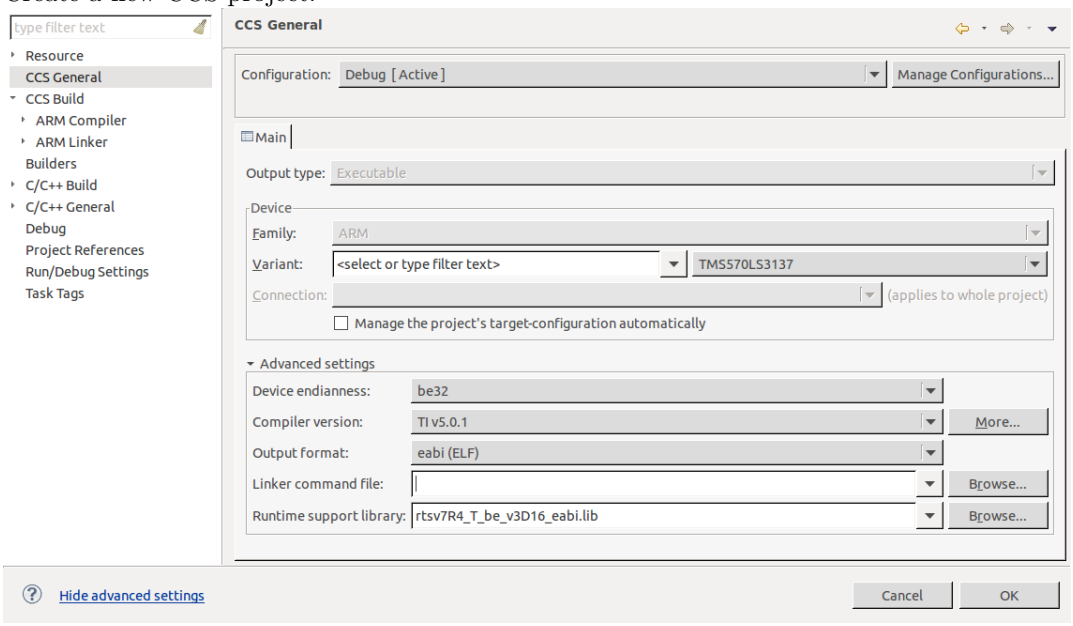
3.5 Base application

In <repo>/rpp/lib/apps/ there is two RPP base applications, `base` and `base_posix`, that already configured for the RPP Library. It is advised that new applications uses this projects a foundations.

To create a new application copy this directory and rename it. Now open files `.project`, `.cproject` and `.ccsproject` (if available) and change any occurrence of the work `base` with the name of your project. Use lower case ASCII letters and underscores only.

Steps to configure a new CCS (ARM, using CGT) RPP application:

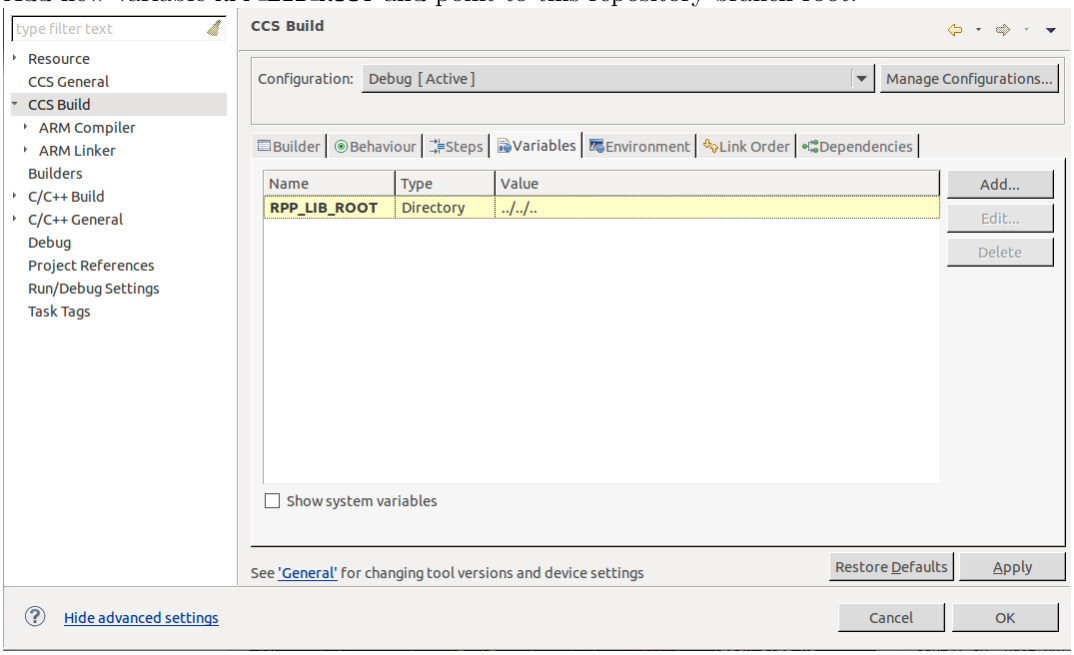
1. Create a new CCS project.



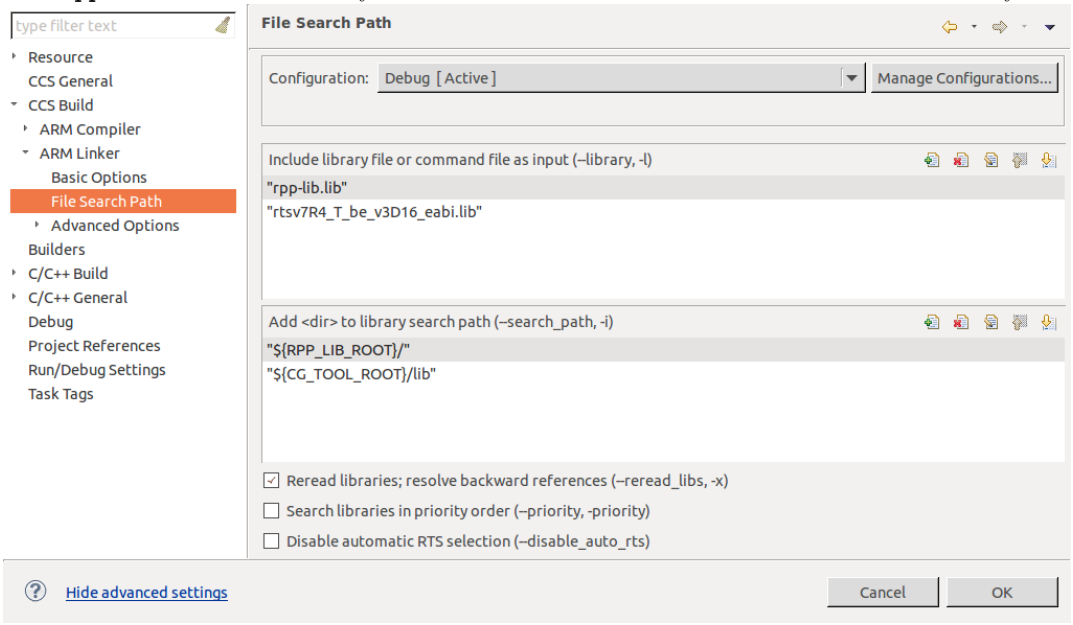
2. Create a normal folder `include`.
3. Create a source folder `src`.
4. Add common `.gitignore` to the root of that project:

```
Debug
Release
.settings/*
```

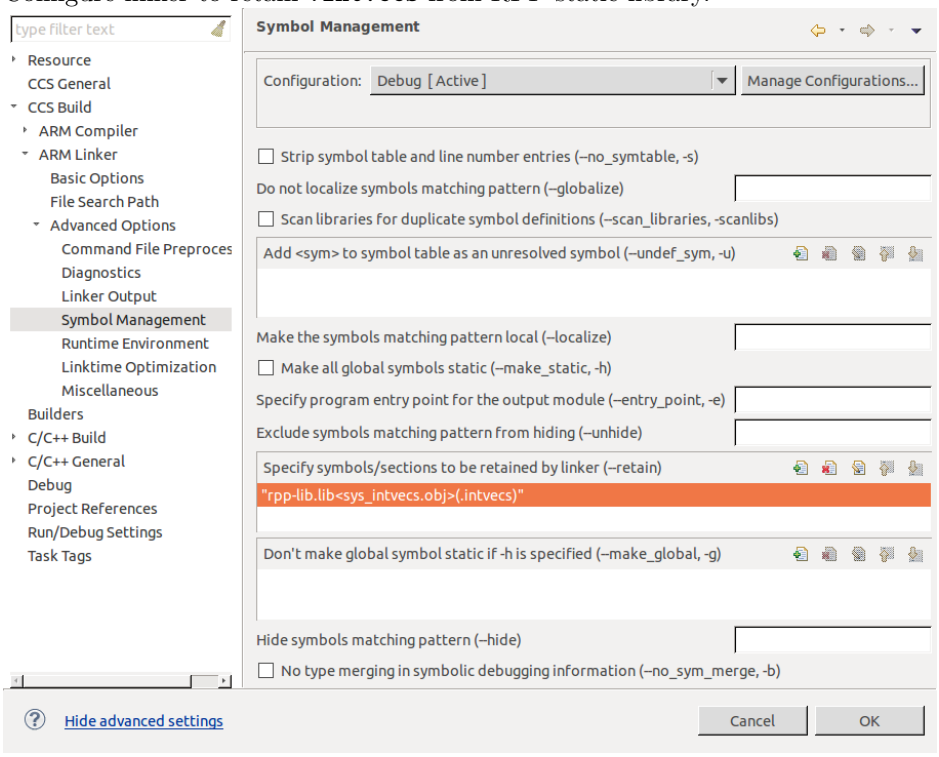
5. Add new variable `RPP_LIB_ROOT` and point to this repository branch root.



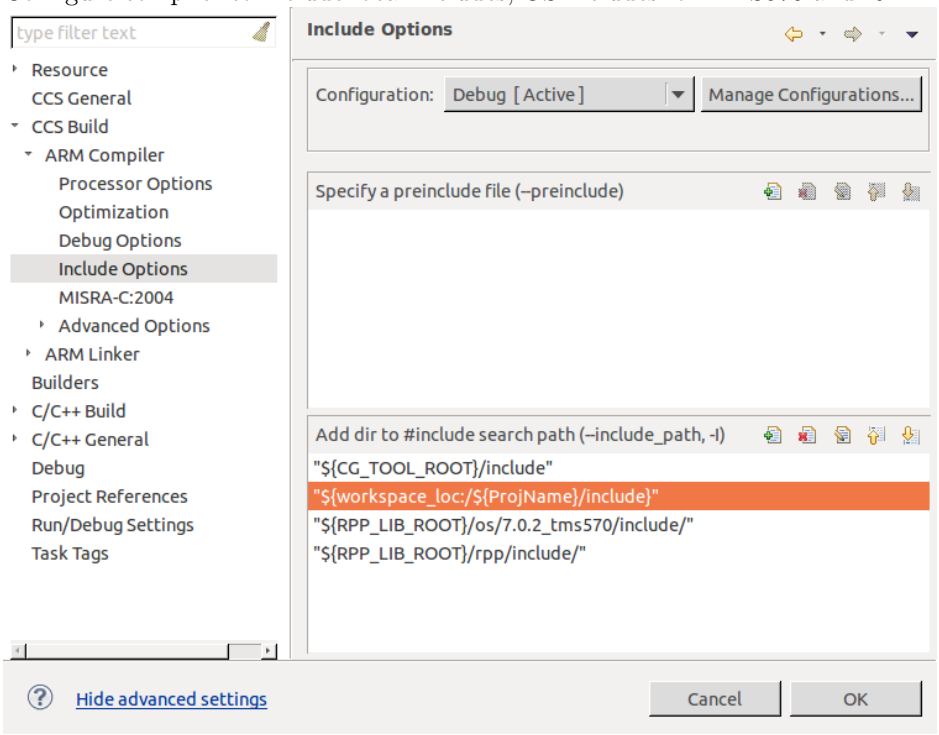
6. Add `rpp-lib.lib` static library to linker libraries and add `RPP_LIB_ROOT` to the library search path.



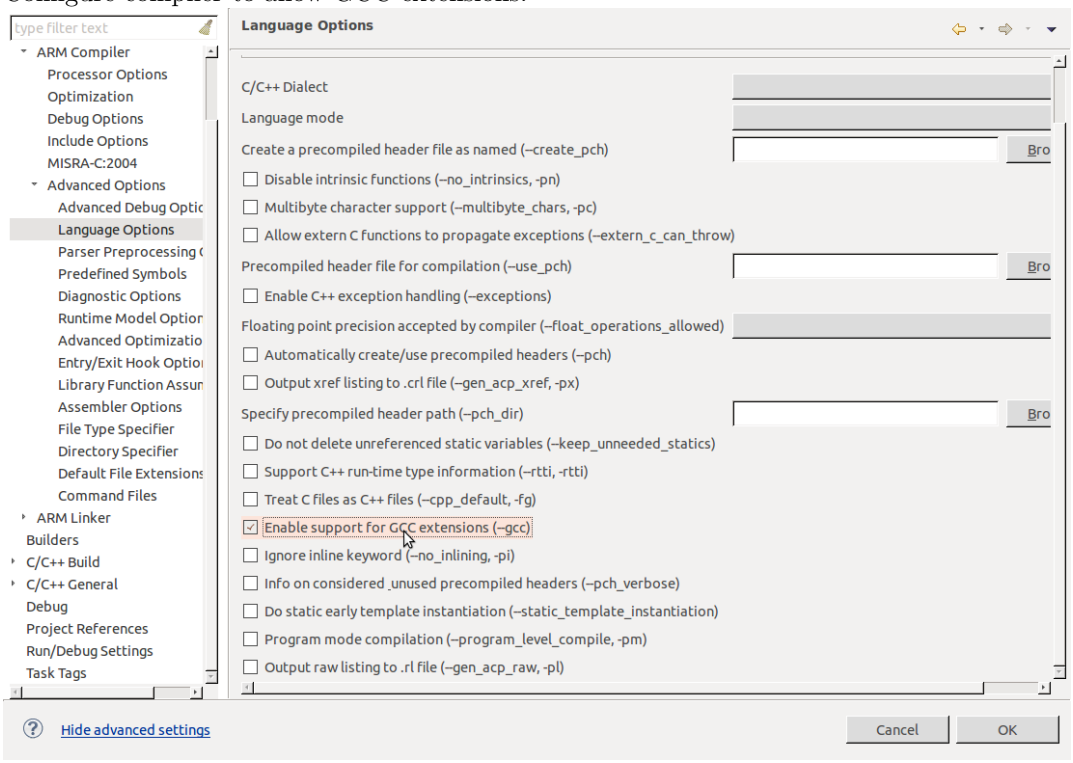
7. Configure linker to retain `.intvecs` from RPP static library.



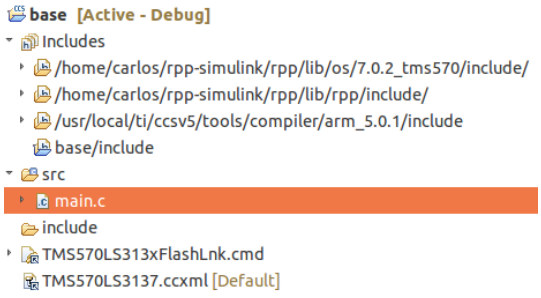
8. Configure compiler to include local includes, OS includes for TMS570 and RPP includes, in that order.



9. Configure compiler to allow GCC extensions.



10. Import and link (do not copy!) linker file and board upload descriptor.

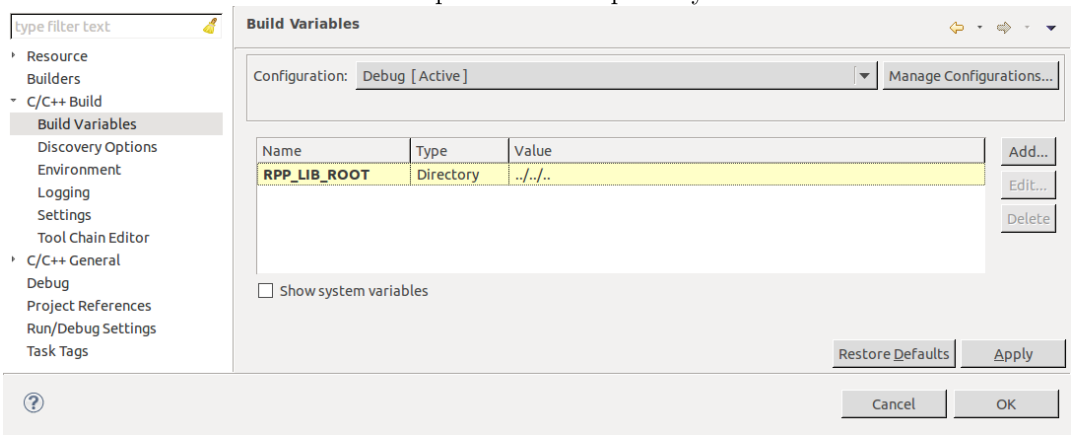


Steps to configure a new GCC (x86(_64)) RPP simulated application:

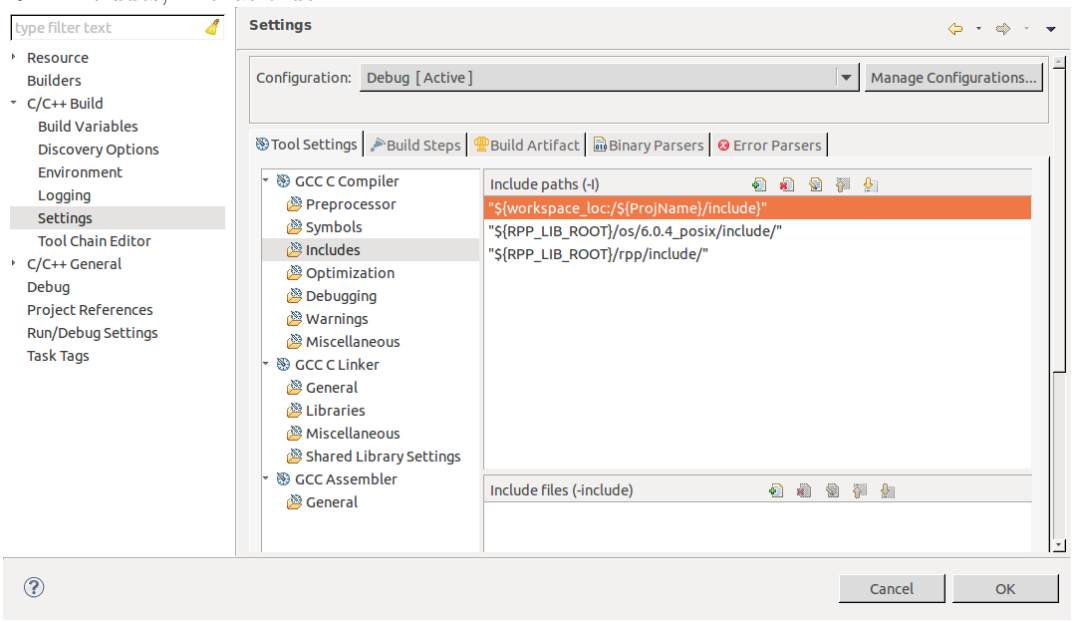
1. Create a new managed C project that uses Linux GCC toolchain.
2. Create a source folder `src`. Link all files from original CCS application to this folder.
3. Create a normal folder `include`. Create a folder `rpp` inside of it.
4. Add common `.gitignore` to the root of that project:

```
Debug
Release
.settings/*
```

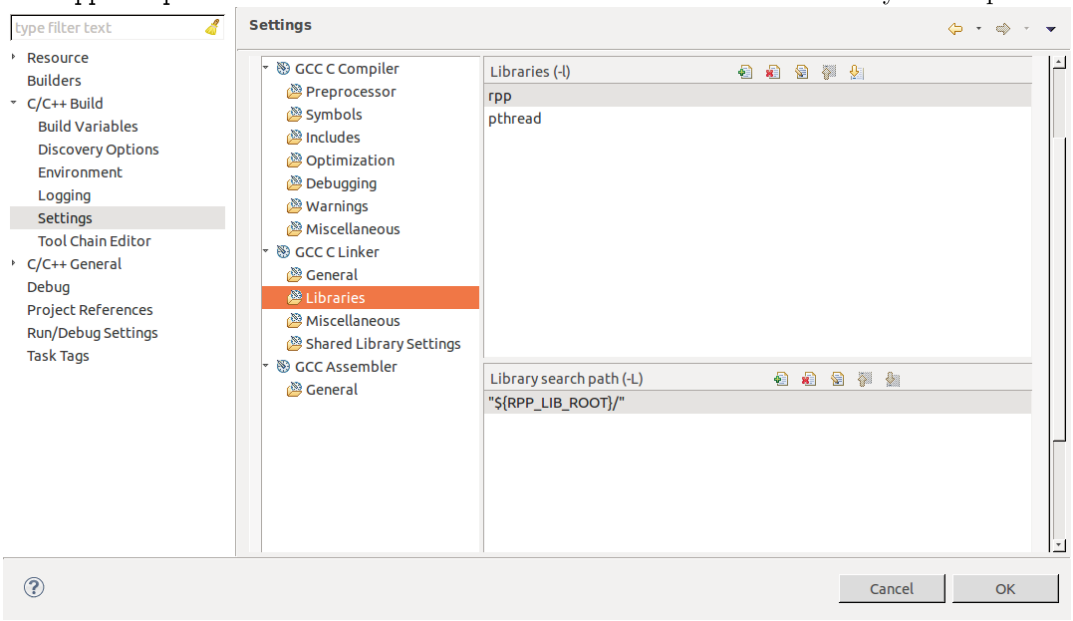
5. Add new variable `RPP_LIB_ROOT` and point to this repository branch root.



6. Configure compiler to include local includes, CCS application includes, OS includes for POSIX and RPP includes, in that order.



7. Add `rpp` and `pthread`to linker libraries and add `RPP_LIB_ROOT` to the library search path.



8. Copy `RppConfig.h` from RPP Library to a new folder `include/rpp` and configure it drop DRV layer dependency: `rppCONFIG_DRV 0`.

In general any RPP application uses the layout/template:

1. Include RPP library header file.

```
1  #include "rpp/rpp.h"
```

2. Create one or as many FreeRTOS task function definitions as required. Those tasks should use functions from this library.

```
void my_task(void* p)
{
    static const portTickType freq_ticks = 1000 / portTICK_RATE_MS;
    portTickType last_wake_time = xTaskGetTickCount();
    while(TRUE) {
        /* Wait until next step */
        vTaskDelayUntil(&last_wake_time, freq_ticks);
        rpp_sci_printf((const char*)"Hello RPP.\r\n");
    }
}
```

3. Create the main function that will:

   - Initialize the RPP board.
   - Spawn the tasks the application requires. Refer to FreeRTOS API for details.
   - Start the FreeRTOS Scheduler. Refer to FreeRTOS API for details.
   - Catch if idle task could not be created. +

```
void main(void)
{
    /* Initialize RPP board */
    rpp_init();

    /* Spawn tasks */
    if(xTaskCreate(my_task, (const signed char*)"my_task",
            512, NULL, 0, NULL) != pdPASS) {
        #ifdef DEBUG
        rpp_sci_printf((const char*)
            "ERROR: Cannot spawn control task.\r\n"
        );
        #endif
        while(TRUE) { asm(" nop"); }
    }

    /* Start the FreeRTOS Scheduler */
    vTaskStartScheduler();

    /* Catch scheduler start error */
    #ifdef DEBUG
    rpp_sci_printf((const char*)
            "ERROR: Problem allocating memory for idle task.\r\n"
        );
    #endif
    while(TRUE) { asm(" nop"); }
}
```

4. Create hook functions for FreeRTOS:

   - `vApplicationMallocFailedHook()` allows to catch memory allocation errors.
   - `vApplicationStackOverflowHook()` allows to catch if a task overflows it's stack. +

```
#if configUSE_MALLOC_FAILED_HOOK == 1
/**
 * FreeRTOS malloc() failed hook.
 */
void vApplicationMallocFailedHook(void) {
    #ifdef DEBUG
    rpp_sci_printf((const char*)
            "ERROR: manual memory allocation failed.\r\n"
        );
    #endif
}
#endif


#if configCHECK_FOR_STACK_OVERFLOW > 0
/**
 * FreeRTOS stack overflow hook.
 */
void vApplicationStackOverflowHook(xTaskHandle xTask,
                                   signed portCHAR *pcTaskName) {
    #ifdef DEBUG
    rpp_sci_printf((const char*)
            "ERROR: Stack overflow : \"%s\".\r\n", pcTaskName
        );
    #endif
}
#endif
```

3.6 API generation

The RPP Layer is formatted using Doxygen documentation generator. This allows to generate a high quality API reference. To generate the API reference do in a terminal:

```
cd <repo>/rpp/lib/rpp/doc/api
doxygen doxygen.conf
xdg-open html/index.html
```

The files under <repo>/rpp/lib/rpp/doc/api/content are used for the API reference generation are their name is self-explanatory:

```
blocks_map.html
blocks.png
cvut.png
footer.html
main_page.dox
```

To install Doxygen see Development environment section.

3.7 API Reference

For the complete API reference please generate the HTML version using the above section instructions. Here is listed the index of functions of each module and their brief.

Please note that not all modules were implemented as part of this project. See RPP Layer Modules for a list of the modules implemented.

3.7.1 DIN API Reference

`int8_t rpp_din_init();`
→ DIN module initialization.

`int8_t rpp_din_ref(uint16_t refA, uint16_t refB);`
→ Configure voltage reference levels for digital inputs using variable reference threshold.

`int8_t rpp_din_setup(uint8_t pin, boolean_t pull_type, boolean_t active, boolean_t can_wake);`
→ Configure given pin.

`int8_t rpp_din_get(uint8_t pin, boolean_t var_thr);`
→ Get the current cached value of the given pin.

`int8_t rpp_din_diag(uint8_t pin);`
→ Get the diagnostic cached value for given pin.

`int8_t rpp_din_update();`
→ Read and update cached values and diagnostic values of all pins. Also commit configuration changes.

3.7.2 LOUT API Reference

`int8_t rpp_lout_init();`
→ LOUT module initialization.

`int8_t rpp_lout_set(uint8_t pin, uint8_t val);`
→ Set the output cache of given pin to given value.

`int8_t rpp_lout_diag(uint8_t pin);`
→ Get the diagnostic cached value for given pin.

`int8_t rpp_lout_update();`
→ Flush cached output values and read back diagnostic values of all pins.

3.7.3 AIN API Reference

`int8_t rpp_ain_init();`
→ AIN module initialization.

`int16_t rpp_ain_get(uint8_t pin);`
→ Get the current analog value on the given pin.

`int8_t rpp_ain_update();`
→ Read and update analog cached values.

3.7.4 AOUT API Reference

`#define RPP_DAC_OA 5.6`
→ DAC output operational amplifier multiplication constant.

`#define RPP_DAC_VREF 2.5`
→ DAC hardware reference voltage.

`int8_t rpp_aout_init();`
→ AOUT module initialization.

`int8_t rpp_aout_setup(uint8_t pin, boolean_t enabled);`
→ Configure enabled/disabled state for given pin.

`int8_t rpp_aout_set(uint8_t pin, uint16_t val);`
→ Set the output cache of given pin to given value.

`int8_t rpp_aout_set_voltage(uint8_t pin, uint16_t mv);`
→ Set output to given voltage.

`int8_t rpp_aout_update();`
→ Flush cached output values and configuration changes.

3.7.5 HBR API Reference

`int8_t rpp_hbr_init();`
→ HBR module initialization.

`int8_t rpp_hbr_enable(int32_t period);`
→ Enable the H-Bridge for control.

`int8_t rpp_hbr_control(double cmd);`
→ Control the H-Bridge direction, enabled/disabled and PWM.

`int8_t rpp_hbr_disable();`
→ Disable the H-Bridge.

3.7.6 MOUT API Reference

`int8_t rpp_hbr_init();`
→ HBR module initialization.

`int8_t rpp_mout_init();`
→ MOUT module initialization.

`int8_t rpp_mout_set(uint8_t pin, uint8_t val);`
→ Set the output of given pin to given value.

`int8_t rpp_mout_get(uint8_t pin);`
→ Get the cached value of the given pin set by rpp_mout_set().

`int8_t rpp_mout_diag(uint8_t pin);`
→ Reads the value on the given diagnostic pin.

3.7.7 HOUT API Reference

`int8_t rpp_hout_init();`
→ HOUT module initialization.

3.7.8 CAN API Reference

`int8_t rpp_can_init();`
→ CAN module initialization.

3.7.9 LIN API Reference

`int8_t rpp_lin_init();`
→ LIN module initialization.

3.7.10 FR API Reference

`int8_t rpp_fr_init();`
→ FR module initialization.

3.7.11 SCI API Reference

`int8_t rpp_sci_init();`
→ SCI module initialization.

`boolean_t rpp_sci_setup(uint32_t baud);`
→ SCI module setup.

`uint16_t rpp_sci_available();`
→ Number of bytes available on input buffer.

`int8_t rpp_sci_read(uint32_t amount, uint8_t* buffer);`
→ Read n number of bytes from input buffer.

`int8_t rpp_sci_read_nb(uint32_t amount, uint8_t* buffer);`
→ Read n number of bytes from input buffer if possible.

`int8_t rpp_sci_write(uint32_t amount, uint8_t* data);`
→ Write n number of bytes to the output buffer.

`int8_t rpp_sci_write_nb(uint32_t amount, uint8_t* data);`
→ Write n number of bytes to the output buffer if possible.

`int8_t rpp_sci_flush(boolean_t buff);`
→ Flush incomming or outgoing buffers.

`int32_t rpp_sci_printf(const char* format, ...);`
→ C style printf using RPP SCI module.

`int8_t rpp_sci_putc(uint8_t byte);`
→ C style putc (put character) using RPP SCI module.

`int16_t rpp_sci_getc();`
→ C style getc (get character) using RPP SCI module.

3.7.12 ETH API Reference

`int8_t rpp_eth_init();`
→ ETH module initialization.

3.7.13 SDC API Reference

`int8_t rpp_sdc_init();`
→ SDC module initialization.

3.7.14 SDR API Reference

`#define RPP_SDR_ADDR_START 0x80000000U`
→ SDRAM start address on RPP board.

`#define RPP_SDR_ADDR_END 0x83FFFFFFU`
→ SDRAM end address on RPP board.

`int8_t rpp_sdr_init();`
→ SDR module initialization.

`int8_t rpp_sdr_setup(boolean_t enable);`
→ Configure SD-RAM logging.

`uint32_t rpp_sdr_available();`
→ Query for the amount of space free on the SD-RAM.

`int32_t rpp_sdr_printf(const char* format, ...);`
→ Store a formatted user string on the log, if logging is enabled.

`int8_t rpp_sdr_clear();`
→ Clear log.

`int8_t rpp_sdr_show(boolean_t start);`
→ Start/Stop the task that sends the log to the SCI.

4 Simulink Coder Target

The Simulink Coder Target allows Simulink model's code generation, compilation and download for the board.

4.1 Description

The Simulink RPP Target provides support for C source code generation from Simulink models and compilation of that code on top of the RPP library and the FreeRTOS operating system. This target uses Texas Instrument ARM compiler (armcl) included in the Code Generation Tools available with Code Composer Studio, and thus it depends on it for proper functioning.

This library also provides support for automatically download the compiled machine code to the RPP board.

4.1.1 Code generation process

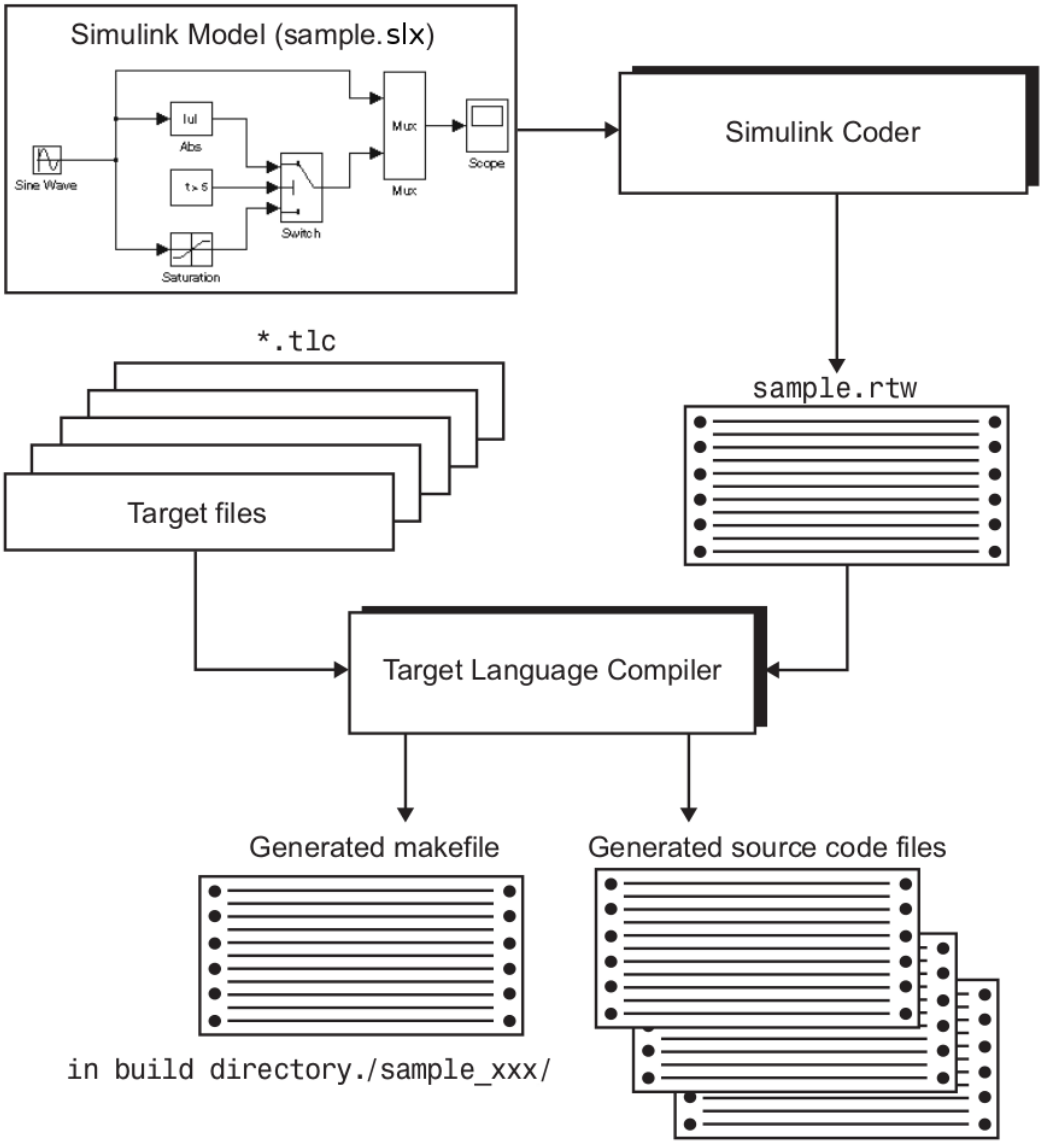


Figure 10: TLC code generation process.

4.2 Subdirectory content description

→ `rpp_setup.m`

RPP Target install script.

This script will, among other things, ask the user to provide the location of the armcl parent directory, infer and save some relevant CCS paths, add paths to Matlab path and build S-Function blocks for user's architecture (using Matlab's mex command line tool).

- Reference:
  - `<repo>/refs/rtw_ug.pdf` p. 1137.

→ `rpp.tlc`

Embedded real-time system target file for RPP.

This file is the system target file (STF), or target manifest file. Functions of the STF include:

- Making the target visible in the System Target File Browser.
- Definition of code generation options for the target (inherited and target-specific).
- Providing an entry point for the top-level control of the TLC code generation process.

- Reference:
  - `<repo>/refs/rtw_ug.pdf` p. 1129 and 1144.

→ `rpp.tmf`

Embedded Coder Template Makefile.

This is just standard Embedded Coder Template Makefile, provided by Matlab. It was slightly modified to support `armcl` particularities and added template rules for assembler files (which were included by the `rpp_lib_support.m` script, but is no longer the case).

- Reference:
  - `<repo>/refs/rtw_ug.pdf` p. 1130 and 1183.

→ `rpp_download.m`

Code download utility for Simulink RPP Target.

This function is optionally executed at the end of the build process if it is successful and the user selected *Download compiled binary to RPP* option on the build configuration panel. This function calls `loadti.sh` script with the generated binary and using configuration for the XDS100v2 JTAG Emulators. The board should be powered and correctly wired.
See Development wiring.

→ `rpp_file_process.tlc`

Code generation custom file processing template.

This file should decide which *main* to generate according to configuration, in particular which mode, Single Tasking or Multitasking, is chosen. The RPP Target ignores this settings because it uses a tasking system based on tasking features provided by FreeRTOS. In consequence is only a wrapper to the *Single Tasking* main, which clearly is not for single tasking.

- Reference:
  - `<repo>/refs/ecoder_ug.pdf` p. 556.
  - `<repo>/refs/ecoder_ref.pdf` p. 1347.

→ `rpp_lib_support.m`

DEPRECATED. Simulink support for RPP library and operating system setup.

This files used to add the source code from the RPP library and operating to the build. This is no longer required when using the static library. This is left for future reference in case new source code needs to be included to the build.

- Reference:
  - `<repo>/refs/rtw_ug.pdf` p. 1058.
  - `<repo>/refs/rtw_ref.pdf` p. 56.

→ `rpp_make_rtw_hook.m`

Build process hooks file.

This file is hook file that invoke target-specific functions or executables at specified points in the build process. In particular, this file handle the copying of required files before the compilation stage.

- Reference:
  - `<repo>/refs/rtw_ug.pdf` p. 1066-1072 and 1131.

→ `rpp_select_callback_handler.m`

RPP Target select callback handler.

This callback function is triggered whenever the user selects the target in the System Target File Browser. Default values for Simulation and configurations parameters are set. Some options are disabled if it is not allowed to be changed by user.

- Reference:
  - `<repo>/refs/rtw_ug.pdf` p. 1211.

→ `rpp_srmain.tlc`

Custom file processing to generate a *main* file.

This file generated the *main* file for the RPP target on top of the RPP library and the FreeRTOS operating system. The `sr` prefix is standard to mark Single Tasking main, which is the case. See `rpp_file_process.m` description above for more information about this.

- Reference:
  - Example in <`matlab`>`/rtw/c/tlc/mw/bareboard_srmain.tlc`.

→ `target_tools.mk`

Makefile for CCS (`armcl`) toolchain support.

This file set variables to CCS tools to support build for this toolchain. This file is included by `rpp.tmf` before declaring the rules for source code.

- Reference:
  - *Include a tool specification settings* comment block in `rpp.tmf`.
  - Compiler options documentation available in `armcl.pdf`.

4.3 Installation procedure

1) Download and install CCS for Linux:

Details on how to setup CCS are available in section TI Code Composer Studio.

2) Install RPP Target:

Open Matlab and type on command window:

```
cd <repo>/rpp/rpp/
rpp_setup()
```

This will launch the RPP setup script. This script will ask the user to provide the path to the CCS compiler root directory (the directory where `armcl` binary is located), normally:

```
<ccs>/tools/compiler/arm_5.X.X/
```

This script will, among other things, ask the user to provide the location of the armcl parent directory, infer and save some relevant CCS paths, add paths to Matlab path and build S-Function blocks for user's architecture (using Matlab's mex command line tool).

3) Create a new model or load a demo:

Demos are located on `<repo>/rpp/demo` or you can start a new model and configure target to RPP. For new models see Target Reference section below.

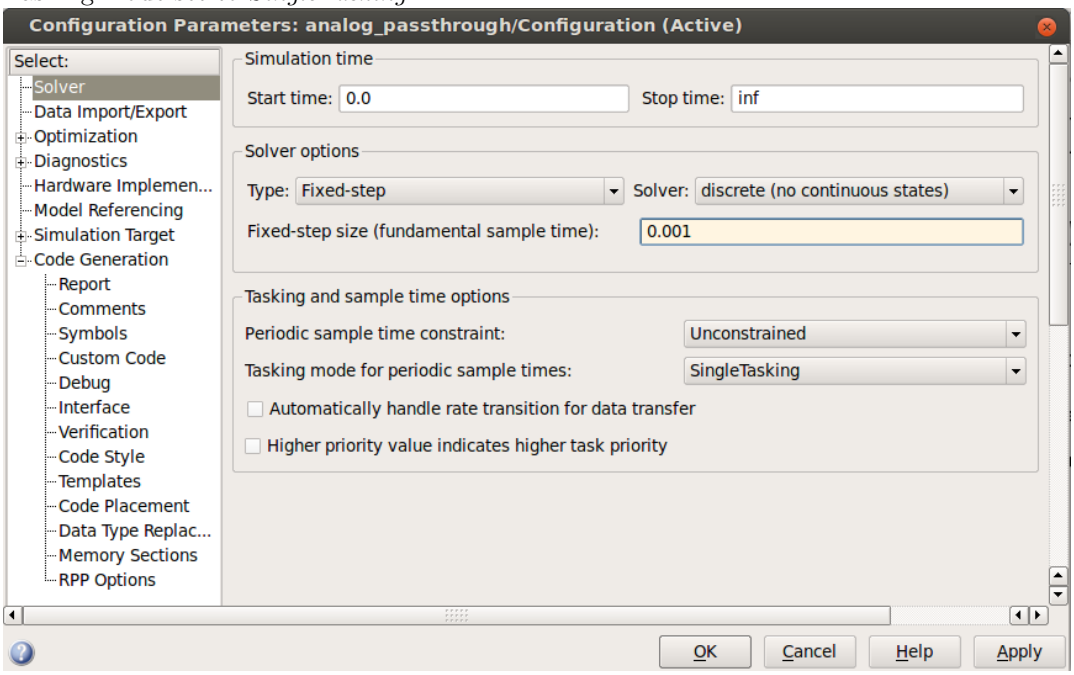
4.4 Target Reference

This section describes the options required or available for running a Simulink model with the RPP Target.

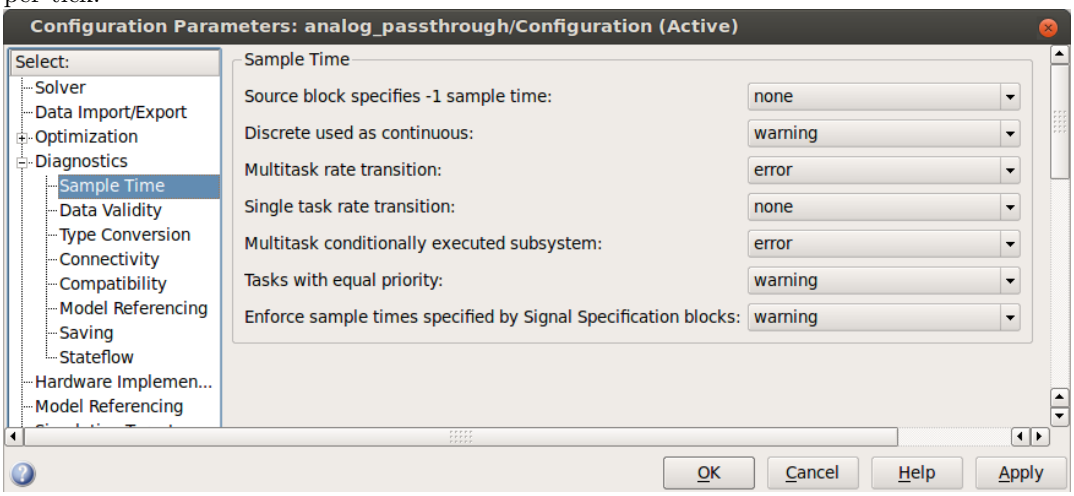
4.4.1 Simulink model options

The Simulink model needs to be configured in the following way:

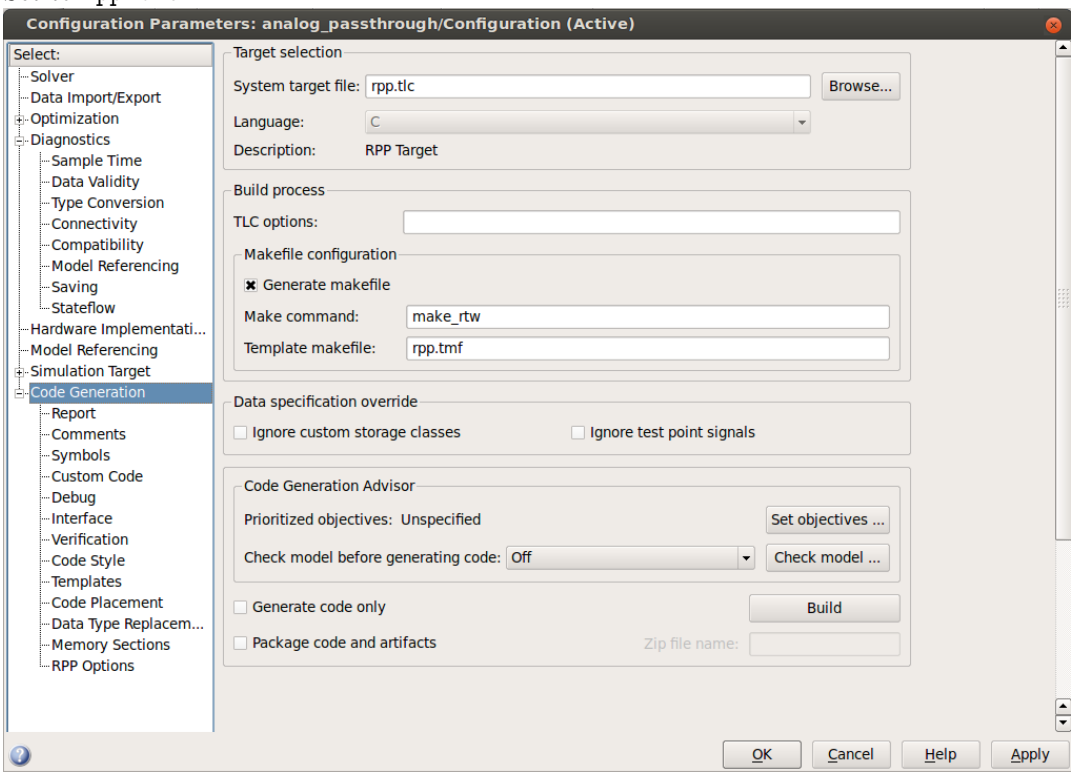
- Solver:
  - *fixed-step discrete*.
  - Tasking mode set to *SingleTasking*.



- Diagnostics - Sample Time:
  - Disable warning source block specifies -1 sampling time. It's ok for the source blocks to run once per tick.



- Code generation:
  - Set to `rpp.tlc`.



Note: Single Tasking is the only currently supported mode. If multitasking is required to be implemented in the future create a new file `rpp_mrmain.tlc` in `<repo>/rpp/rpp/` and edit `rpp_file_process.tlc` to use that file instead when multitasking is selected.

4.4.2 RPP Target options

The RPP Target include the following configuration options, all of them configurable per model under `Code Generation` → `RPP Options`:

- **C system stack size**: this parameter is passed directly to the linker for the allocation of the stack. Note that this is the stack for the application when running outside a FreeRTOS task, normally before the scheduler has started and for system routines. Default value is 4096.
- **C system heap size**: this parameter is passed directly to the linker for the allocation of the heap. See Appendix A: Notes on FreeRTOS memory management for an important information about this parameter.
- **Model step task stack size**: this parameter will be passed to the `xTaskCreate()` that creates the task for the model to run. In a Simulink model there is always two tasks:
  - The worker task. This task is the one that executes the model step. This task requires enough stack memory to execute the step. Take into account for example than only a single call to `rpp_sci_printf()` requires, with current configuration, 128 bytes from the stack. This value should be minor than the C system heap and leaving enough heap for the system tasks. See Appendix A: Notes on FreeRTOS memory management for more information.
  - The control task. This task controls when the worker task should execute and controls overruns.
- **Download compiled binary to RPP**: if set, this option will download the generated binary to the board after the model is successfully built. Note that this option is unaware of the option *Generate code only* in the *Code Generation* options panel, so it will try to upload even if only source code has being generated, failing graciously or uploading an old binary laying around in the build directory. This option calls the `rpp_download.m` script, which is in turn a wrapper on the `loadti.sh` script. More information on the `loadti.sh` script can be found in:

  ```
  <css>/ccs_base/scripting/examples/loadti/readme.txt
  http://processors.wiki.ti.com/index.php/Loadti
  ```

  The `loadti.sh` script will close after the download of the generated program and in consequence the execution of the loaded program will stop (because it work as the CCS debug server). In order to test the loaded model a manual reset of the board is always required after a successful download.
- **Print model metadata to SCI at start**: if set this option will print a message to the Serial Communication Interface when the model start execution on the board. This is very helpful to identify the model running on the board. The message is in the form:

  ```
  'model_name' - generated_date (TLC tlc_version)
  ```

  For example:

  ```
  'hbridge_analog_control' - Wed Jun 19 14:10:44 2013 (TLC 8.3 (Jul 20 2012))
  ```

5 Simulink Block Library

The Simulink Block Library is a set of blocks that allows Simulink models to use board IO and communication peripherals.

5.1 Description

As part of this project the ideal set was defined, but not all blocks were implemented. The following table shows the current status of the block library.

CATEGORY	NAME	STATUS*	MNEMONIC	LRH*
System blocks	Configuration block	X	[CONF]	RppConfig.h
Logic IO blocks	Digital Input block	T	[DIN]	rpp_din.h
	Digital Output block	T	[LOUT]	rpp_lout.h
	Analog Input block	T	[AIN]	rpp_ain.h
	Analog Output block	T	[AOUT]	rpp_aout.h
Power output blocks	H-Bridge Control block	T	[HBR]	rpp_hbr.h
	Power output block	T	[MOUT]	rpp_mout.h
	High-Power output block	X	[HOUT]	rpp_hout.h
Communication blocks	CAN Bus receive block	X	[CANR]	rpp_can.h
	CAN Bus send msg block	X	[CANS]	- Idem -
	LIN receive block	X	[LINR]	rpp_lin.h
	LIN send msg block	X	[LINS]	- Idem -
	FlexRay receive block	X	[FRR]	rpp_fr.h
	FlexRay send msg block	X	[FRS]	- Idem -
	SCI receive block	T	[SCIR]	rpp_sci.h
	SCI send msg block	T	[SCIS]	- Idem -
	SCI configure block	T	[SCIC]	- Idem -
	Ethernet receive block	X	[ETHR]	rpp_eth.h
	Ethernet send msg block	X	[ETHS]	- Idem -
Logging/Storage blocks	SD Card write block	T	[SDCW]	rpp_sdc.h
	SDRAM write block	X	[SDRW]	rpp_sdr.h
Trigger blocks	Overrun detected block	X	[TROR]	- None -
	Stack overflow detected block	X	[TRSO]	- None -
	Malloc Failed detected block	X	[TRMF]	- None -

Legend:

- *LRH : Library Reference Header.
- *STATUS :
  - X - Unimplemented. Files non present.
  - P - Unimplemented. Files present.
  - W - Work in progress.
  - I - Implemented.
  - T - Implemented and tested.

Notes:

Each block that can detect fault condition should have a trigger output.
High-power output provides current flow as an input to the model.

5.1.1 C MEX S-Functions

All of the blocks are implemented as a C Mex S-Function coded by hand. In the this section the approach taken is explained.

C-MEX S-Function:

- C : Implemented in C language. Other options are Fortran and Matlab language itself.
- MEX: Matlab Executable. They are compiled by Matlab GCC wrapper called MEX.
- S-Function: System Function, as opposed to standard functions, or user functions.

A C-MEX S-Function is a structured C file that includes the following mandatory callbacks:

1. `mdlInitializeSizes`:
   Specify the number of inputs, outputs, states, parameters, and other characteristics of the C MEX S-function.
2. `mdlInitializeSampleTimes`:
   Specify the sample rates at which this C MEX S-function operates.
3. `mdlOutputs`:
   Compute the signals that this block emits.
4. `mdlTerminate`:
   Perform any actions required at termination of the simulation.

Plus many more optional callbacks. Relevant optional callbacks are:

1. `mdlCheckParameters`:
   Check the validity of a C MEX S-function's parameters.
2. `mdlRTW`:
   Generate code generation data for a C MEX S-function.
3. `mdlSetWorkWidths`:
   Specify the sizes of the work vectors and create the run-time parameters required by the C MEX S-function.
4. `mdlStart`:
   Initialize the state vectors of the C MEX S-function.

A complete list of callbacks can be found in:

http://www.mathworks.com/help/simulink/create-cc-s-functions.html

The way a C-MEX S-Function participates in a Simulink simulation is shown by the following diagram:

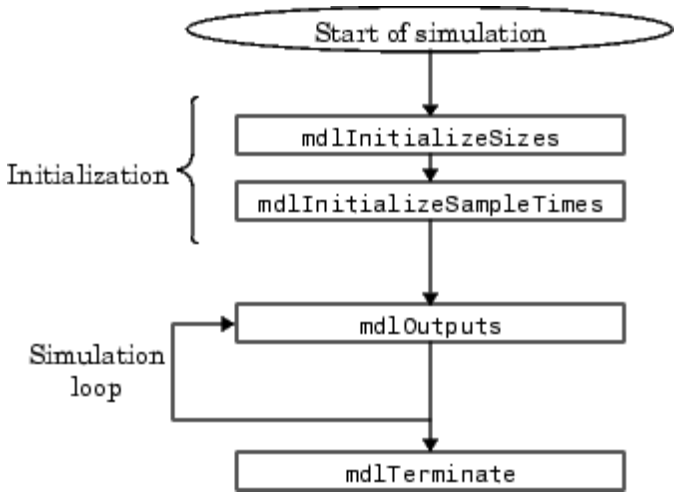


Figure 11: Simulation cycle of a S-Function.

In general, a S-Function can perform calculations and inputs and outputs for simulation. Because the blocks implemented for this project are for hardware peripherals control and IO the blocks are implemented as pure sink or pure source. That is, the S-Function is a descriptor of the block but does not any calculation, input or output for simulation.

The S-Functions required could be implemented in several ways:

1. Writing the S-Function.
   Using this method, the user hand write a new C S-Function and associated TLC file. This method requires the most knowledge about the structure of a C S-Function.
2. Using an S-Function Builder block.
   Using this method, the user enter the characteristics of the S-function into a block dialog. This method does not require any knowledge about writing S-Functions. However, a basic understanding of the structure of an S-Function can make the S-Function Builder dialog box easier to use.
3. Using the Legacy Code Tool (LCT).
   Using this command line method, the user define the characteristics of your S-function in a data structure in the MATLAB workspace. This method requires the least amount of knowledge about S-Functions.

From the above, the LCT is a tool that can be called within Matlab workshop that allows to generate source code for S-Functions given the descriptor of a C function call. This approach is used by most of the other targets reviewed for this project. The descriptor is a Matlab file with definitions like the following:

```
%% GPIO Write
% Populate legacy_code structure with information
GPIOWrite = legacy_code('initialize');
GPIOWrite.SFunctionName = 'sfun_GPIOWrite';
GPIOWrite.HeaderFiles = {'gpiolct.h'};
GPIOWrite.SourceFiles = {'gpiolct.c'};
GPIOWrite.OutputFcnSpec = 'GPIOWrite(uint32 p1, uint8 u1, uint8 u2)';
% Support calling from within For-Each subsystem
GPIOWrite.Options.supportsMultipleExecInstances = true;
```

The interface and implementation files specified should hold the declaration and implementation of the `OutputFcnSpec` function. This tool will generate a simple S-Function that will input and output the values required by that function. This approach was **not** for this project, mainly because:

- The RPP Library requires that after some actions (like setting one LOUT output) the changes are committed to the hardware, or before some other actions (like getting the value from DIN using the fixed threshold) the values cached are updated. And the implementation of a wrapper function that would update or commit the changes wasn't considered because of the efficiency impact it would have.
- Furthermore, the error handling of the function call is not considered, and for some blocks (like MOUT and HOUT) the diagnostic handling is mandatory.
- Also, the dialog parameters of the S-Function cannot be validated otherwise than data type (cannot validate range, for example).
- For future improvements the LCT cannot generate code for simulation, and a lot of S-Function options cannot not be fine tuned.
- Finally, the generated code is very obscure, hard to read and to maintain in case the above functionality had to be implemented on top of the generated code.

Similarly the hand written S-Functions shares a large amount of code like parameters scalar, data type and range validation, standard options for this kind of blocks, unused functions, among other. Because of this a mini framework for writing S-Functions for RPP was implemented in the form of two files that are directly included at the beginning and end of the S-Function implementation: `header.c` and `trailer.c`.

This mini-framework reduces the amount of required code for each S-Function considerably, making easier to maintain and adapt. Because each S-Function is a program by itself there is no need to use interface files and the files are directly included.

The final form of the S-Function is a C file of around 100 lines of code with the following layout:

- Define S-Function name `S_FUNCTION_NAME`.
- Include header file `header.c`.
- In `mdlInitializeSizes` define:
  - Number of *dialog* parameter.
  - Number of input ports.
    - Data type of each input port.
  - Number of output ports.
    - Data type of each output port.
  - Standard options for driver blocks.
- In `mdlCheckParameters`:
  - Check data type of each parameter.
  - Check range, if applicable, of each parameter.
- In `mdlSetWorkWidths`:
  - Map *dialog* parameter to *runtime* parameters.
    - Data type of each *runtime* parameter.
- Define symbols for unused functions.
- Include trailer file `trailer.c`.

The C-MEX S-Function implemented can be compile with the following command:

```
<matlabroot>/bin/mex sfunction_{mnemonic}.c
```

As noted the standard is to always prefix S-Function with `sfunction_` and use lower case mnemonic of the block.

Also a script called `compile_blocks.m` is included that allows all `sfunctions_*.c` to be fed to the `mex` compiler so all S-Functions are compiled at once. To use this script, in Matlab do:

```
cd <repo>/rpp/blocks/
compile_blocks()
```

5.1.2 Target Language Compiler files

C code generated from a Simulink model is placed on a file called <modelname>.c along with other support files in a folder called <modelname>_<target>/. For example, the source code generated for model foobar will be placed in current Matlab directory foobar_rpp/foobar.c.

The file <modelname>.c has 3 main functions:

- void <modelname>_step(void):
  This function recalculates all the outputs of the blocks and should be called once per step. This is the main working function.
- void <modelname>_initialize(void):
  This function is called only once before the first step is issued. Default values for blocks IOs should be placed here.
- void <modelname>_terminate(void):
  This function is called when terminating the model. This should be used to free memory of revert other operations made on the initialization function. With current implementation this function should never be called unless an errors is detected and in most models it is empty.

In order to generate code for each one of those functions each S-Function implement a TLC file for *inlining* the S-Function on the generated code. The TLC files are files that describe how to generate code for a specific C-MEX S-Function block. They are programmed using TLC own language and include C code within TLC instructions, just like LaTeX files include normal text in between LaTeX macros.

TLC files are located under <repo>/rpp/blocks/tlc_c/ directory. For a diagram on how TLC files work see Code generation process section.

The standard for a TLC file is to be located under the tlc_c subfolder from where the S-Function is located and to use the very exact file name as the S-Function but with the .tlc extension:

sfunction_foo.c → tlc_c/sfunction_foo.tlc

The TLC files implemented for this project use 3 hook functions in particular (other are available, see TLC reference documentation):

- BlockTypeSetup:
  BlockTypeSetup executes once per block type before code generation begins. This function can be used to include elements required by this block type, like includes or definitions.
- Start:
  Code here will be placed in the void <modelname>_initialize(void). Code placed here will execute only once.
- Outputs:
  Code here will be placed in the void <modelname>_step(void) function. Should be used to get the inputs o a block and/or to set the outputs of that block.

The general layout of the TLC files implemented for this project are:

- In `BlockTypeSetup`:
  Call common function `%<RppCommonBlockTypeSetup(block, system)>` that will include the `rpp/rpp.h` header file (can be called multiple times but header is included only once).
- `Start`:
  Call setup routines from RPP Layer for the specific block type, like HBR enable, DIN pin setup, AOUT value initialization, SCI baud rate setup, among others.
- `Outputs`:
  Call common IO routines from RPP Layer, like DIN read, AOUT set, etc. Success of this functions is checked and in case of failure error is reported to the block using ErrFlag.

5.2 Subdirectory content description

→ `header.c` and `trailer.c`

RPP framework for simple S-Functions.

This files are included at the head and tail of each S-Function file. They include refactored and commonly repeated structures that pollute S-Functions implementations. They include basic includes, required definitions, macro definitions, common functions implementations and documentation on optional functions and commented prototypes for optional model calls/hooks.

- Reference:
  - See header of those files.

→ `sfunction_{mnemonic}.c`

C-MEX S-Function implementation for {mnemonic} block.

This file implements the {mnemonic} block using C-MEX S-Function API. See the reference for information about the S-Function API.

- Reference:
  - `<repo>/refs/sfunctions.pdf`

→ `tlc_c/sfunction_{mnemonic}.tlc`

Target Language Compiler (TLC) file for {mnemonic} block.

This file implements the C code inlining for {mnemonic} block. See the reference for information about the TLC API.

- Reference:
  - `<repo>/refs/rtw_tlc.pdf`

→ `tlc_c/common.tlc`

Common TLC functions.

This file implements common TLC functions used by all the blocks.

- Reference:
  - None.

→ `slblocks.m`

Simulink library control file.

This file allows a group of blocks to be integrated into the Simulink Library and Simulink Library Browser. This file is required by Simulink in order to interpret this folder as a block library. For information about this file see the references.

- Reference:
  - `<repo>/refs/rtw_ug.pdf` p. 1127

→ `rpp_lib.slx`

RPP Simulink block library.

Simulink block library that includes all the blocks. This file is referenced by `slblocks.m`

- Reference:
  - None.

→ compile_blocks.m

Blocks compilation script.

This script compiles all the sfunction blocks to MEX executables. This script is called by the `rpp_setup()` function in order make all the blocks available to the Simulink environment or it can be called independently when developing S-Functions.

- Reference:
  - None.

5.3 Block Library Reference

This section describes each one of the Simulink blocks implements as part of this project:

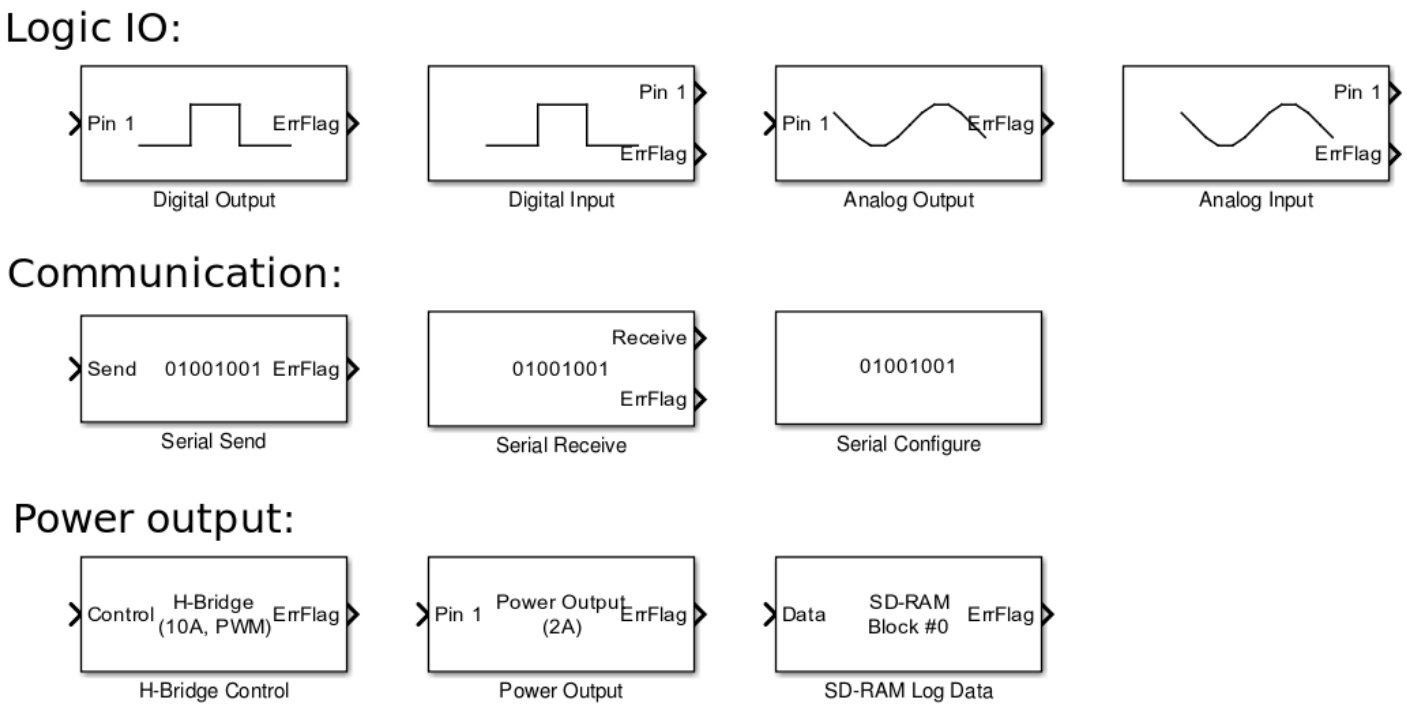


Figure 12: Simulink RPP Block Library.

5.3.1 DIN Digital Input block

```
Inputs      : 0
    None

Outputs     : 2
    bool    Digital Input
    bool    ErrFlag

Parameters  : 2
    uint8   Pin number [1-16]
    bool    Use variable threshold
```

This block allows to read the digital inputs on the RPP board. The variable threshold check change the read mode of the pin. The ErrFlag should raise if `rpp_din_update()` or `rpp_din_get()` returns error. `rpp_din_update()` is called just by the first DIN block in the model and thus only the first block could raise the flag because of this. In case an errors occurs the return value will always be LOW (0). Because the ErrFlag should never set, once set the following steps will never clear it back.

- **Tested**:
  - Changing the pin.
  - Compilation and general use.
  - Using variable threshold.
- **Untested**:
  - Faulty situation for the ErrFlag to set.
- **Not working**:

RPP API functions used:

- `rpp_din_setup()`.
- `rpp_din_update()`.
- `rpp_din_get()`.

Relevant demos:

- `digital_passthrough`.
- `hbridge_digital_control`.

5.3.2 LOUT Digital Output block

```
Inputs      : 1
    bool    Digital Output

Outputs     : 1
    bool    ErrFlag

Parameters  : 1
    uint8   Pin number [1-8]
```

This block allows to write to the digital outputs on the RPP board. The ErrFlag should raise if `rpp_lout_set()` or `rpp_lout_update()` returns error. Because the ErrFlag should never set, once set the following steps will never clear it back. `rpp_lout_update()` is called on each block, which is not the most efficient but guaranties consistent behavior.

Status:

- **Tested**:
  - Changing the pin.
  - Compilation and general use.
- **Untested**:
  - Faulty situation for the ErrFlag to set.
- **Not working**:

RPP API functions used:

- `rpp_lout_set()`.
- `rpp_lout_update()`.

Relevant demos:

- `digital_passthrough`.
- `led_blink_all`.
- `led_blink`.

5.3.3 AIN Analog Input block

```
Inputs      : 0
    None

Outputs     : 2
    uint16  Analog Input
    bool    ErrFlag

Parameters  : 1
    uint8   Pin number [1-12]
```

This block allows to read the analog inputs on the RPP board. The ErrFlag should if raise `rpp_ain_update()` or `rpp_ain_get()` returns error. `rpp_ain_update()` is called just by the first DIN block in the model and thus only the first block could raise the flag because of this. In case an errors occurs the return value will always be 0. Because the ErrFlag should never set, once set the following steps will never clear it back.

Status:

- **Tested**:
  - Changing the pin.
  - Compilation and general use.
- **Untested**:
  - Faulty situation for the ErrFlag to set.
- **Not working**:

RPP API functions used:

- `rpp_ain_update()`.
- `rpp_ain_get()`.

Relevant demos:

- `analog_passthrough`.
- `hbridge_analog_control`.
- `log_analog_input`.

5.3.4 AOUT Analog Output block

```
Inputs      : 1
    uint16  Analog Output

Outputs     : 1
    bool    ErrFlag

Parameters  : 1
    uint8   Pin number [1-4]
    bool    UseVoltage
```

This block allows to write to the analog outputs on the RPP board. The UseVoltage flag allows the user to configure if block inputs should be interpreted as raw DAC value or millivolts. The ErrFlag should raise if `rpp_aout_update()` or `rpp_aout_set()` (or `rpp_aout_set_voltage()` depending on block configuration) returns error. Because the ErrFlag should never set, once set the following steps will never clear it back.

`rpp_aout_update()` is called on each block but the implementation provides this to be efficient.

There is a know bug on the RPP Library, check `rpp_aout_update()` on the RPP API for details. Because of this, the outputs of the DACs are initialized on the first step of the model and not on the model initialization.

Status:

- **Tested**:
  - Changing the pin.
  - Changing voltage/value flag.
  - Compilation and general use.
- **Untested**:
  - Faulty situation for the ErrFlag to set.
- **Not working**:
  - Initializing DACs on model's initialization.

RPP API functions used:

- `rpp_aout_setup()`.
- `rpp_aout_set()`, or
- `rpp_aout_set_voltage()`.
- `rpp_aout_update()`.

Relevant demos:

- `analog_passthrough`.
- `analog_sinewave`.

5.3.5 HBR H-Bridge Control block

```
Inputs      : 1
    double  Control

Outputs     : 1
    bool    ErrFlag

Parameters  : 0
    None
```

This block allows to control the H-Bridge on the RPP board. The ErrFlag should raise only if `rpp_hbr_control()` returns error. The H-Bridge is initialized with the default frequency (˜18kHz). A future improvement could include a parameter to set the frequency. Because the ErrFlag should never set, once set the following steps will never clear it back.

Status:

- **Tested**:
  - Compilation and general use.
- **Untested**:
  - Faulty situation for the ErrFlag to set.
- **Not working**:

RPP API functions used:

- `rpp_hbr_enable()`.
- `rpp_hbr_control()`.

Relevant demos:

- `hbridge_analog_control`.
- `hbridge_digital_control`.
- `hbridge_sinewave_control`.

5.3.6 MOUT Power Output block

```
Inputs      : 1
    bool    Power Output

Outputs     : 1
    bool    ErrFlag

Parameters  : 1
    uint8   Pin number [1-6]
```

This block allows to write the power outputs (2A) on the RPP board. The ErrFlag should raise only if `rpp_mout_set()`returns error. Note that `rpp_mout_set()` returns error only if some bad parameter or in case it could detect a faulty condition on the pin in a very very short period of time after setting the value, see the function API for details. If the faulty condition persist on the next step the call will successfully detect the faulty condition and ErrFlag should set. Because the ErrFlag should never set, once set the following steps will never clear it back.

Status:

- **Tested**:
  - Changing the pin.
  - Compilation and general use.
- **Untested**:
  - Faulty situation for the ErrFlag to set.
- **Not working**:

RPP API functions used:

- `rpp_mout_set()`.

Relevant demos:

- `power_toggle`.

5.3.7 SCIR Serial Comm. Interface Receive

```
Inputs      : 0
    None

Outputs     : 2
    uint8   Data
    bool    ErrFlag

Parameters  : 0
    None
```

This block allows to receive a byte from the SCI. The ErrFlag should raise if `rpp_sci_read_nb()` doesn't succeed. The behavior of the ErrFlag is different from others blocks in that this block will set or clear the flag if the call fails of success at each step. Note that this block uses the non-blocking call to read the SCI and thus will never cause an overrun.

Status:

- **Tested**:
  - Receiving data.
  - Compilation and general use.
  - Faulty situation for the ErrFlag to set.
- **Untested**:
- **Not working**:

RPP API functions used:

- `rpp_sci_read_nb()`.

Relevant demos:

- `echo_char`.

5.3.8 SCIS Serial Comm. Interface Send

```
Inputs      : 1
    uint8   Data

Outputs     : 1
    bool    ErrFlag

Parameters  : 2
    bool    UsePrintf
    string  PrintFormat [SETTING]
```

This block allows to send a byte to the SCI or to print a formatted string that uses that byte. The UsePrintf flag allows to user to select `rpp_sci_write_nb()` (raw send) or `rpp_sci_printf()` (formatted print) as the function the block should use on code generation. If UsePrintf is set the PrintFormat string parameters SETTING is used as the format specifier. Note that this value is inserted raw between quotes on code generation and thus there is no validation on it. User should always put any valid integer specifier for the value on the input of the block.

The behavior of this block depends if UsePrintf is set or not. If set, the call `rpp_sci_printf()` (a blocking call) could potentially overrun the step. Also, the ErrFlag will set only if `rpp_sci_printf()` returns an error, and because it should never set, once set it will never clear back. On the contrary, if UsePrintf is clear, the call `rpp_sci_write_nb()` (non-blocking) is used and thus the step cannot be overrun, but because is a best-effort call it cannot guarantee that all the data will be sent. In the case that not all data could be sent, the ErrFlag will set, but it will clear back if the next step is able to send all it's data (which with the current implementation is just one byte).

A possible future improvement for this block is to allow input to be non-scalar so user can print a whole string in one step using raw non-blocking write. This is currently possible if input configuration is adapted in S-Function and TLC. The problem this could pose is is that for printf user should include specifiers for all the cells in the non-scalar input, and if unknown, then printf cannot be used.

Status:

- **Tested**:
  - Sending data.
  - Compilation and general use.
- **Untested**:
  - Faulty situation for the ErrFlag to set.
- **Not working**:

RPP API functions used:

- `rpp_sci_write_nb()`, or `rpp_sci_printf()`.

Relevant demos:

- `echo_char` and `hello_world`.

5.3.9 SCIC Serial Comm. Interface Configure

```
Inputs      : 0
    None

Outputs     : 0
    None

Parameters  : 1
    uint32  Baud rate
```

This block allows to configure the baud rate of the SCI. There should only one block of this type per model, and this requirement is not validated, but the inclusion of several blocks is harmless and will just produce the baud rate to be changed several times, being the final baud rate to be the one of the last executed block. This block just executes on model initialization and not on each step.

Status:

- **Tested**:
  - Changing baud rate.
  - Compilation and general use.
- **Untested**:
  - Using more than one block in a model.
- **Not working**:

RPP API functions used:

- `rpp_sci_setup()`.

Relevant demos:

- `echo_char`.
- `hello_world`.

5.3.10 SDRW SD-RAM Write

```
Inputs      : 1
    double  Data

Outputs     : 1
    bool    ErrFlag

Parameters  : 2
    uint8   Block ID
    string  PrintFormat [SETTING]
```

This block allows to log a double value to the SD-RAM. User needs to provide a valid PrintFormat string to format and register the double value on the log. The PrintFormat string should include two specifiers:

- For the block ID. Any valid integer specifier.
- For the value to log. Any valid double specifier.

Note that the value of PrintFormat is inserted raw between quotes on code generation and thus there is no validation on it. Error to provide a valid PrintFormat could generate compilation errors on even runtime errors (normally this generates a warning on compile time). Note that the function for logging used is `rpp_sdr_printf()`, which is a blocking call, and can potentially overrun the step. The ErrFlag will set if `rpp_sdr_printf()` returns an error (for example out of memory), but will clear back if the next step the call to this function is successful.

Status:

- **Tested**:
  - Logging data.
  - Compilation and general use.
- **Untested**:
  - Faulty situation for the ErrFlag to set.
- **Not working**:

RPP API functions used:

- `rpp_sdr_printf()`.

Relevant demos:

- `log_analog_input`.

6 Simulink Demos Library

The Simulink RPP Demo Library is a set of Simulink models that use blocks from the Simulink RPP Block Library and generates code using the Simulink RPP Target.

6.1 Description

This demos library is used as a test suite for the Simulink RPP Block Library but they are also intended to show basic programs built using it. Because of this, the demos try to use more than one type of block and more than one block per block type.

The following table shows the current status of the demos:

Name	**Implemented**	**Tested**
analog_passthrough	YES	SUCCESS
analog_sinewave	YES	SUCCESS
digital_passthrough	YES	SUCCESS
echo_char	YES	SUCCESS
hbridge_analog_control	YES	SUCCESS
hbridge_digital_control	YES	SUCCESS
hbridge_sinewave_control	YES	SUCCESS
hello_world	YES	SUCCESS
led_blink_all	YES	SUCCESS
led_blink	YES	SUCCESS
log_analog_input	YES	SUCCESS
power_toggle	YES	SUCCESS

In the reference below you can find a complete description for each of the demos.

6.2 Subdirectory content description

→ `{demo}.slx`

A Simulink demo.

This subdirectory just includes all the Simulink demos described in the following section.

6.3 Demos Reference

This section describes the demos implemented as part of this project that uses the Simulink RRP Block Library and generates code using the RPP Simulink Target.

6.3.1 Analog pass-through

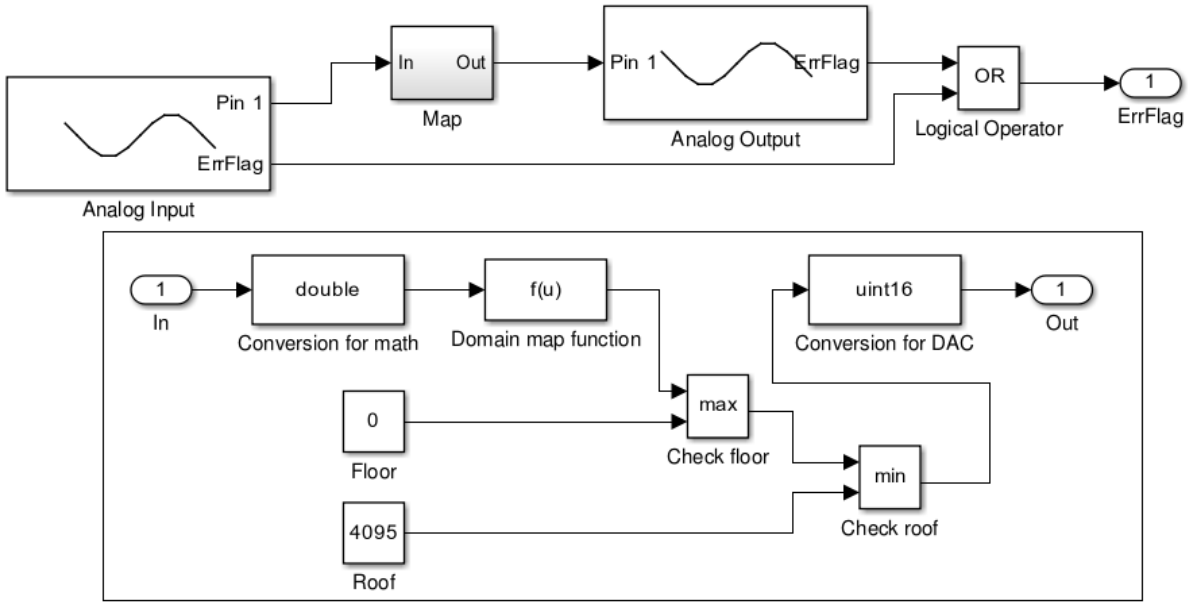


Figure 13: Analog Passthrough Simulink demo for RPP.

Description:

This demo will read analog input 1 and write it to analog output 1.

In laboratory the minimum read value for analog input a 0 volts is 107. The maximum read at 12 volts is 2478. The map subsystem will map the input domain (AIN) `[110, 2400]` to the output domain (AOUT) `[0, 4095]`.

6.3.2 Analog sinewave

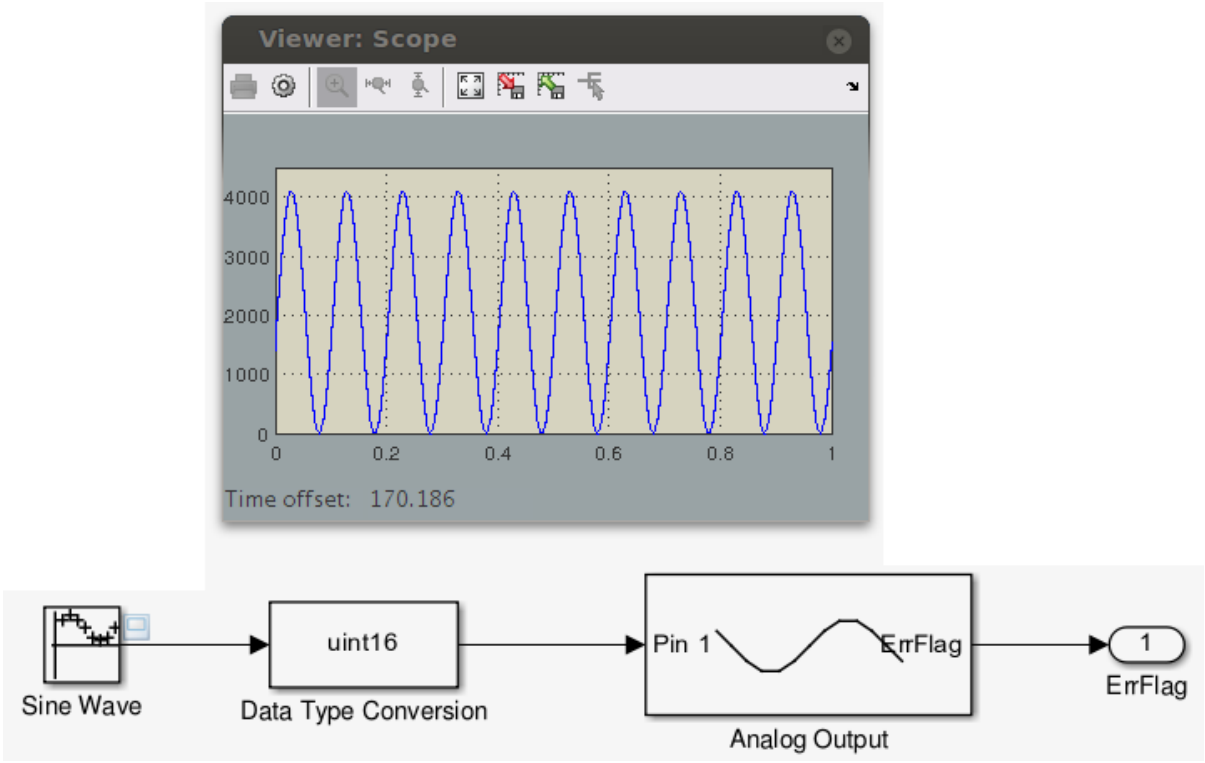


Figure 14: Analog Sinewave Simulink demo for RPP.

Description:

This demo will generate a sinewave on analog output 1. Siwave is 10Hz and sampling rate is set to 1000Hz (driven from Simulink step of 1ms, same as operating system). Amplitude is set to use AOUT full range [0-4095] which means output amplitude will be [0-12] volts.

The Software oscilloscope shown should match an external one connected to AOUT 1.

Note that the driver configuration of the MCP4922 is set to unbuffered (which should eventually be changed to buffered) and thus the last resolution millivolts are lost.

6.3.3 Digital pass-through

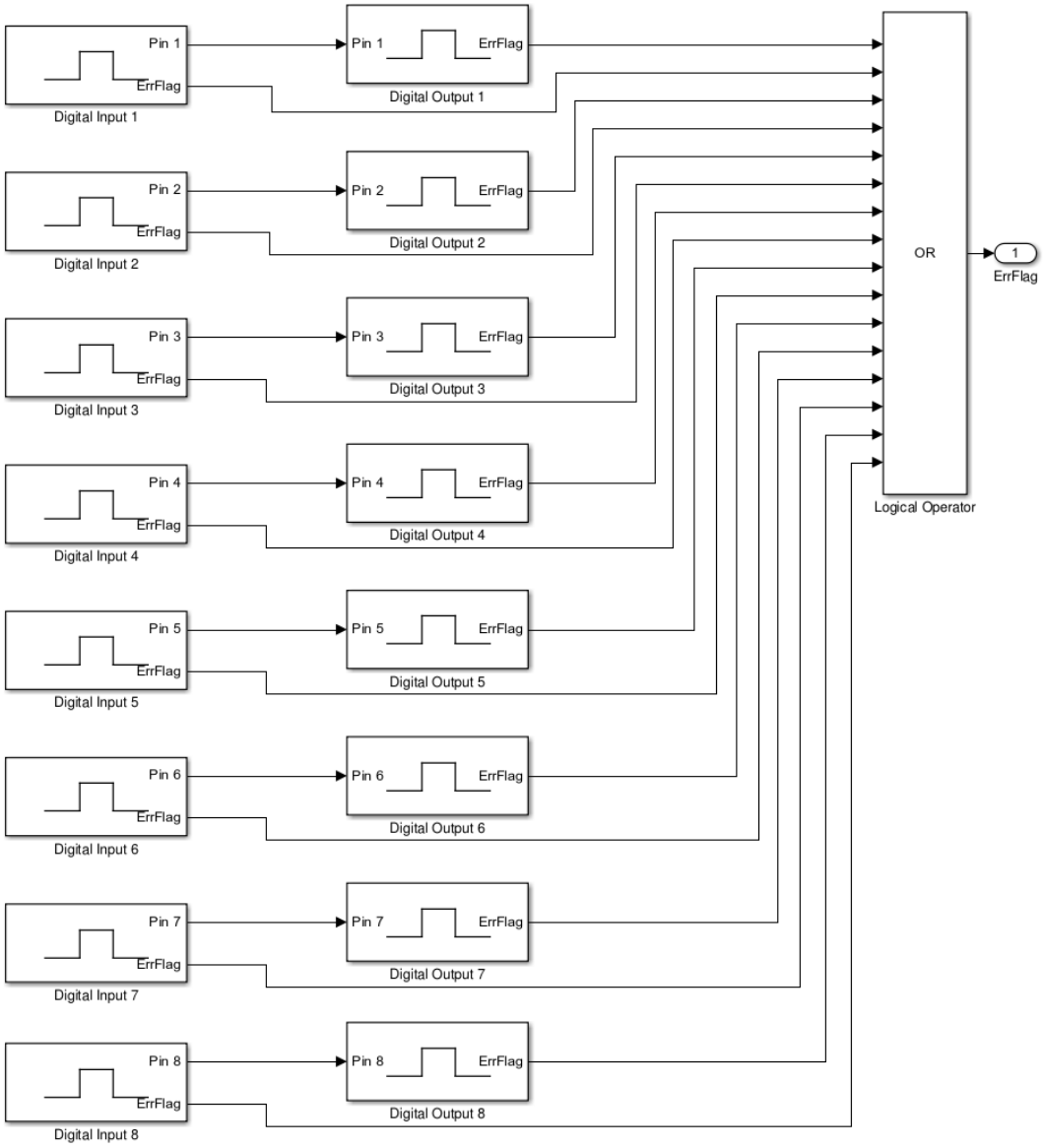


Figure 15: Digital Pass-through Simulink demo for RPP.

Description:

This demo will directly pass the digital values read on DIN [1-8] to LOUT [1-8], and thus acting as a digital pass-through or gateway.

Also note that all the ErrFlag are aggregated on a global ErrFlag.

6.3.4 Echo char

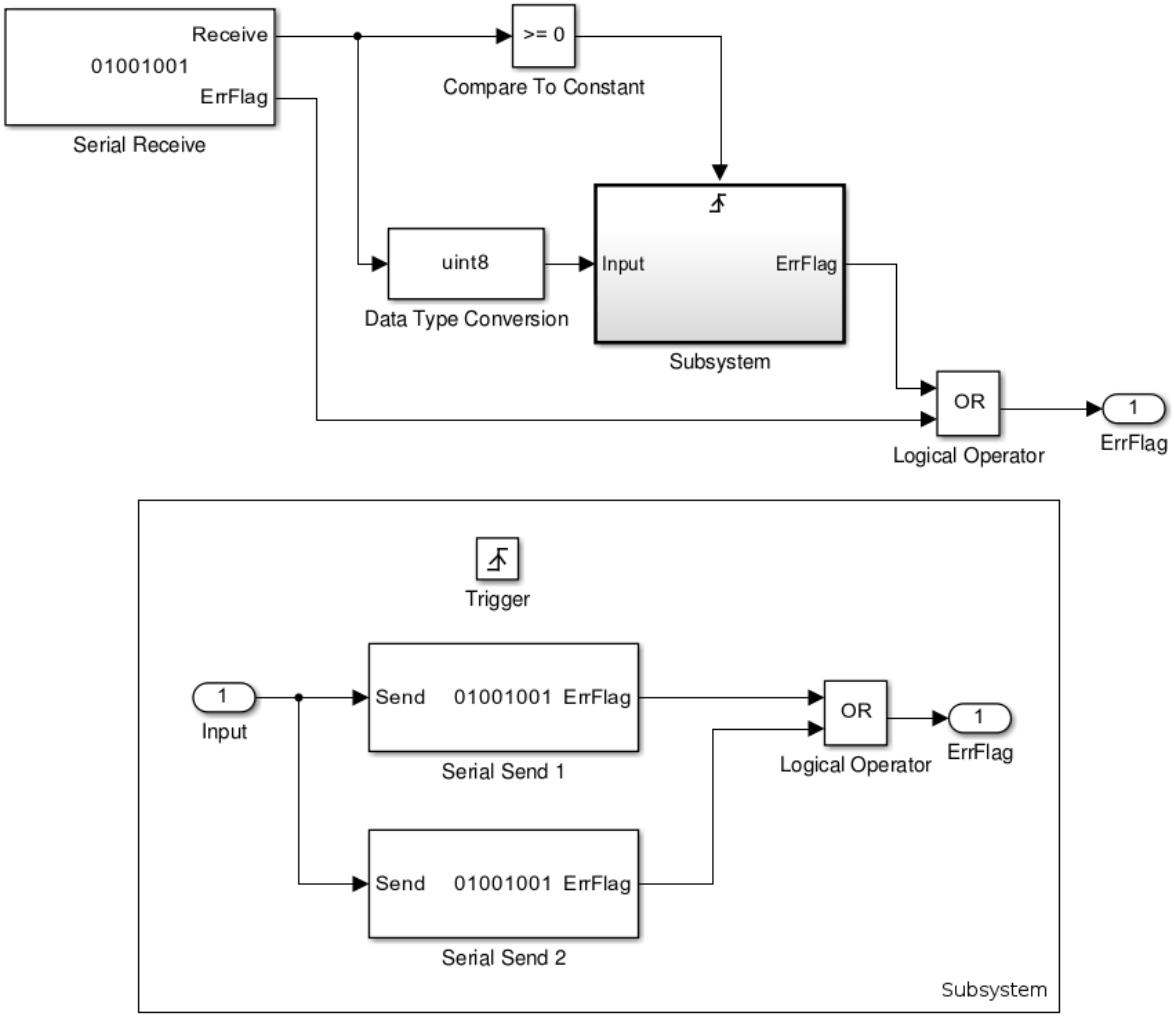


Figure 16: Echo Character Simulink demo for RPP.

Description:

This demo will echo twice (print back) any character received through the Serial Communication Interface (9600-8-N-1).

Note that the send subsystem is implemented a as *triggered* subsystem and will execute only if data is received, that is, Serial Receive output is non-negative. Negative values are errors.

6.3.5 H-bridge analog control

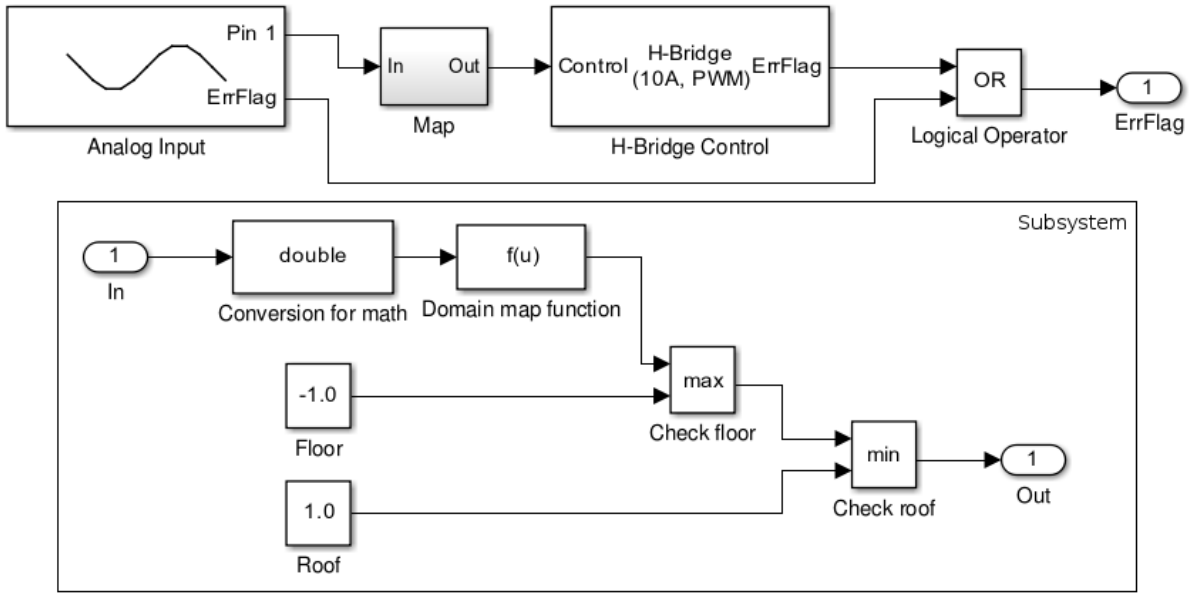


Figure 17: H-Bridge Analog Control Simulink demo for RPP.

Description:

This demo will read values from the analog input, map them, and control the H-Bridge. This allows a motor connected to the H-Bridge to be controlled with a potentiometer connected to Analog Input 1.

Setting the potentiometer to output around 6 volts will stop the motor. Less (or greater) than 6 volts will trigger the motor in one sense (or in the other sense) and speed proportional with 1% resolution.

In laboratory the minimum read value for analog input is 107 at 0 volts. The maximum read at 12 volts is 2478. The map subsystem will map the input domain (AIN)`[110, 2400]` to the output domain (HBR)`[-1.0, 1.0]`.

6.3.6 H-bridge digital control

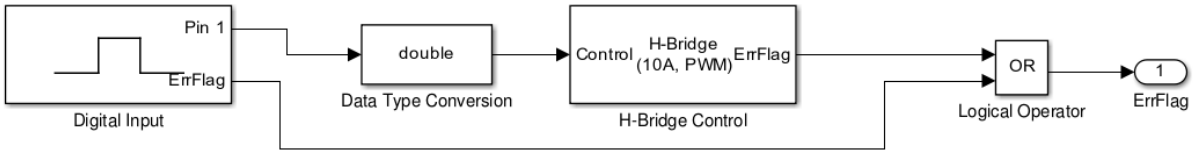


Figure 18: H-Bridge Digital Control Simulink demo for RPP.

Description:

This demo toggle the H-Bridge from stop to full speed in one direction using digital input 1. So basically is a ON/OFF switch on DIN 1 for a motor connected on the HBR. Note the data type conversion because the output of the DIN is a boolean and the input to the HBR is a double.

6.3.7 H-bridge sine wave control

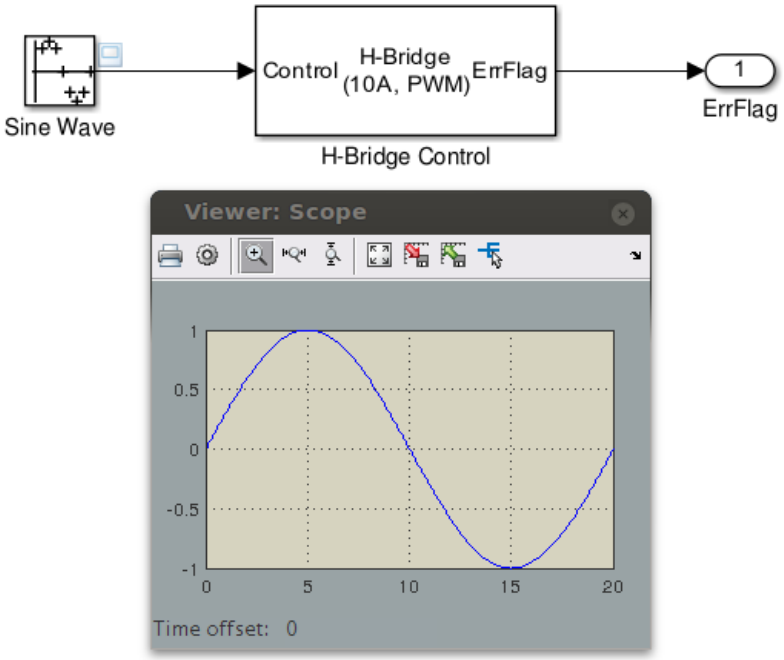


Figure 19: H-Bridge Sinewave Control Simulink demo for RPP.

Description:

This demo will generate a sine wave to control the H-Bridge. Sine wave is one period per 20 seconds or 0.05Hz. Sampling rate is 20Hz or 100 samples per 1/4 of period (for 1% speed resolution change).

Note that the Software oscilloscope should is not the output of the H-Bridge, the H-Bridge will change current sense and the duty cycle of the pulse that drive it (PWM), it does not output analog values. The Software oscilloscope just shows what the input to the HBR block is.

6.3.8 Hello world

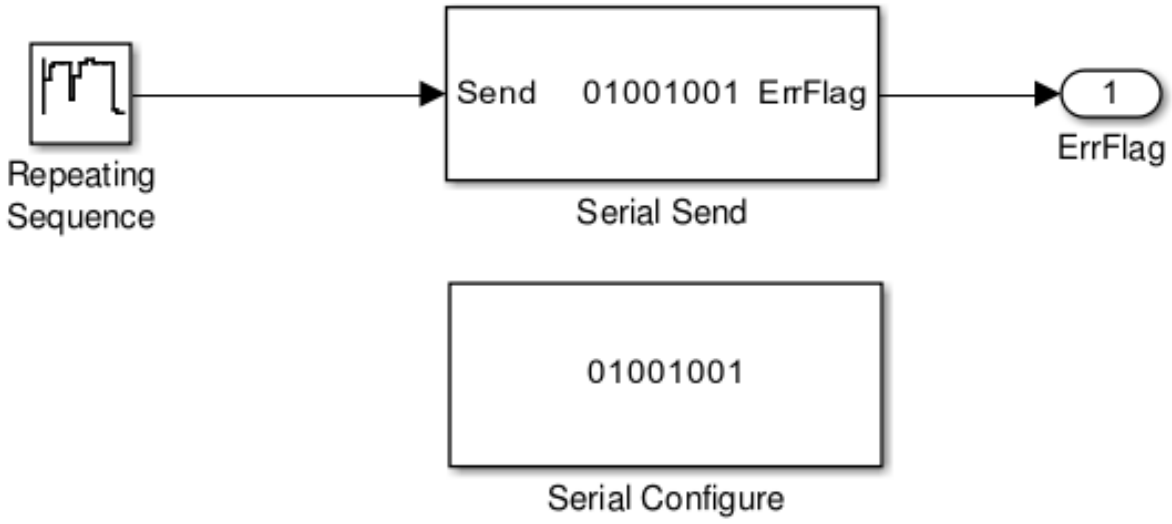


Figure 20: Hello World Simulink demo for RPP.

Description:

This demo will print `"Hello Simulink"` to the Serial Communication Interface (9600-8-N-1) one character per second. The output speed is driven by the Simulink model step which is set to one second.

6.3.9 LED blink

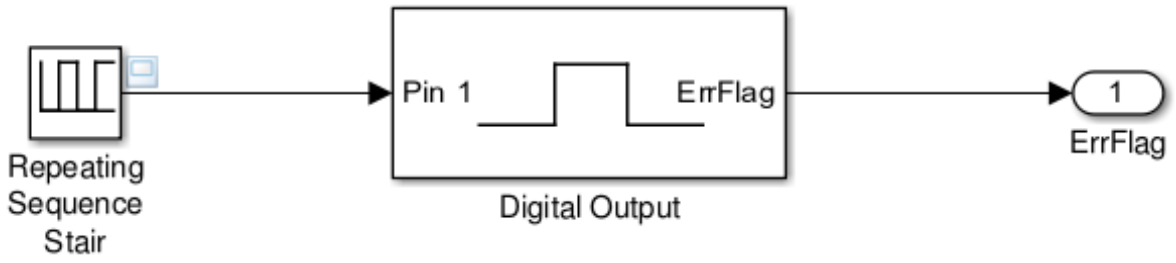


Figure 21: LED Blink Simulink demo for RPP.

Description:

This the simplest demo of all that shows the basics of using the RPP target and blocks. The goal of this demo is to show the configuration of the model (not shown on the picture above), that is, how the RPP Simulink Coder Target is setup, general model setup and step setup.

This demo will toggle each second a LED connected on LOUT 1. The timing is set by the Simulink model step which is set to 1 second.

6.3.10 LED blink all

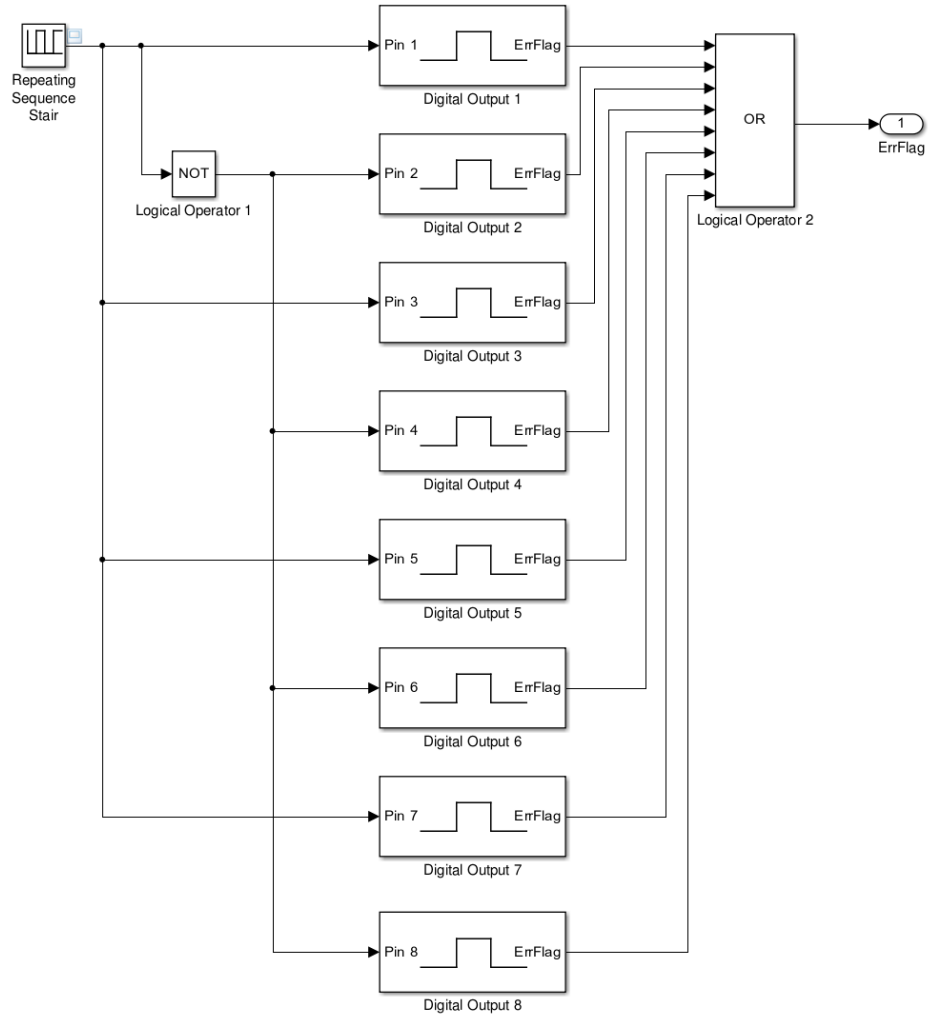


Figure 22: LED Blink All Simulink demo for RPP.

Description:

This demo will toggle all LEDs connected to the LOUT port. Even outputs pins will be negated. Toggle will happen each second. The timing is driven by Simulink model step configuration that is set to 1 second. All blocks ErrFlags are aggregated into one global ErrFlag.

6.3.11 Log analog input

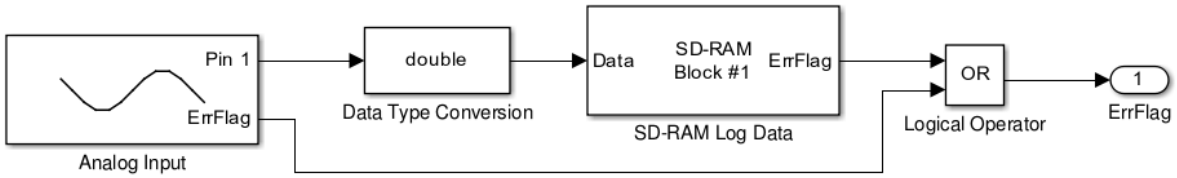


Figure 23: Log Analog Input Simulink demo for RPP.

Description:

This demo will log once per second the value read on the analog input 1. User can read the log using the SCI logging integrated command processor (9600-8-N-1). Logging block ID set to 1. The timing is driven by Simulink model step configuration that is set to 1 second.

6.3.12 Power toggle

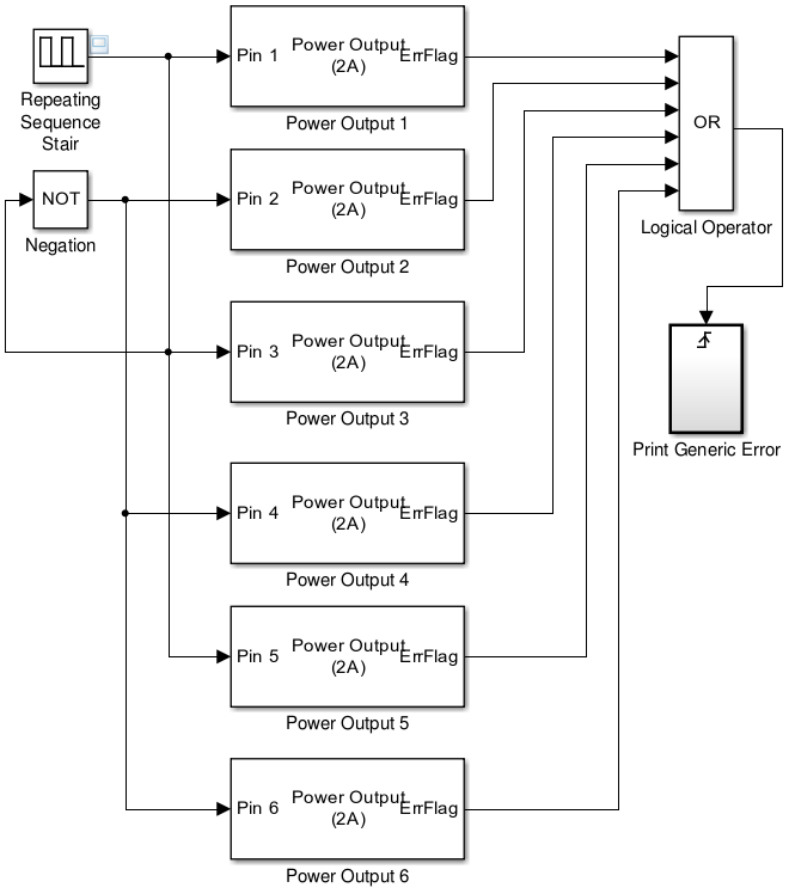


Figure 24: Power Toggle Simulink demo for RPP.

Description:

This demo will toggle the power output once per second. If an error is detected on at least one of the outputs a generic error message is printed to the serial line. The timing is driven by Simulink model step configuration that is set to 1 second. Power outputs can drive a load up to 2A, so please take into account required safety considerations.

7 Glossary

ADC *Analog to Digital Converter.*
Hardware circuitry that converts a continuous physical quantity (usually voltage) to a digital number that represents the quantity's amplitude.

AIN *Analog Input.*
Mnemonic to refer to or something related to the analog input (ADC) hardware module.

AOUT *Analog Output.*
Mnemonic to refer to or something related to the analog output (DAC) hardware module.

CAN *Controller Area Network.*
The CAN Bus is a vehicle bus standard designed to allow microcontrollers and devices to communicate with each other within a vehicle without a host computer. In this project it is also used as mnemonic to refer to or something related to the CAN hardware module.

CGT *Code Generation Tools.*
Name given to the tool set produced by Texas Instruments used to compile, link, optimize, assemble, archive, among others. In this project is normally used as synonym for "Texas Instruments ARM compiler and linker."

DAC *Digital to Analog Converter.*
Hardware circuitry that converts a digital (usually binary) code to an analog signal (current, voltage, or electric charge).

DIN *Digital Input.*
Mnemonic to refer to or something related to the digital input hardware module.

ECU *Engine Control Unit.*
A type of electronic control unit that controls a series of actuators on an internal combustion engine to ensure the optimum running.

ETH *Ethernet.*
Mnemonic to refer to or something related to the Ethernet hardware module.

FR *FlexRay.*
FlexRay is an automotive network communications protocol developed to govern on-board automotive computing. In this project it is also used as mnemonic to refer to or something related to the FlexRay hardware module.

GPIO *General Purpose Input/Output.*
Generic pin on a chip whose behavior (including whether it is an input or output pin) can be controlled (programmed) by the user at run time.

HBR *H-Bridge.*
Mnemonic to refer to or something related to the H-Bridge hardware module. A H-Bridge is an electronic circuit that enables a voltage to be applied across a load in either direction.

HOUT *High-Power Output.*
Mnemonic to refer to or something related to the 10A, PWM, with current sensing, high-power output hardware module.

IDE *Integrated Development Environment.*
An IDE is a Software application that provides comprehensive facilities to computer programmers for software development.

LCT *Legacy Code Tool.*
Matlab tool that allows to generate source code for S-Functions given the descriptor of a C function call.

LIN *Local Interconnect Network.*
The LIN is a serial network protocol used for communication between components in vehicles. In this project it is also used as mnemonic to refer to or something related to the LIN hardware module.

LOUT *Logic Output.*
Mnemonic to refer to or something related to the digital output hardware module. It is logic output (100mA), as opposed to power outputs (2A, 10A).

MBD *Model-Based Design.*
Model-Based Design (MBD) is a mathematical and visual method of addressing problems associated with designing complex control, signal processing and communication systems.

MEX *Matlab Executable.*
Type of binary executable that can be called within Matlab. In this document the common term used is 'C MEX S-Function", which means Matlab executable written in C that implements a system function.

MOUT *(Motor) Power Output.*
Mnemonic to refer to or something related to the 2A push/pull power output hardware module.

PWM *Pulse-width modulation.*
Technique for getting analog results with digital means. Digital control is used to create a square wave, a signal switched between on and off. This on-off pattern can simulate voltages in between full on and off by changing the portion of the time the signal spends on versus the time that the signal spends off. The duration of "on time" is called the pulse width or *duty cycle*.

RPP *Rapid Prototyping Platform.*
Name of the automotive hardware board. Also generic term to define something related to the board, like the RPP Library, RPP Layer, RPP API, etc.

SCI *Serial Communication Interface.*
Serial Interface for communication through hardware's UART using communication standard RS-232. In this project it is also used as mnemonic to refer to or something related to the Serial Communication Interface hardware module.

SDC *SD-Card.*
Mnemonic to refer to or something related to the SD-Card hardware module.

SDR *SD-RAM.*
Mnemonic to refer to or something related to the SD-RAM hardware module for logging.

TLC *Target Language Compiler.*
Technology and language used to generate code in Matlab/Simulink.

UART *Universal Asynchronous Receiver/Transmitter.*
Hardware circuitry that translates data between parallel and serial forms.

8 References

- Horn, M. (2013). *Software obsluhující periferie a flexray na automobilové řídicí jednotce.* (Unpublished master's thesis, Czech Technical University in Prague, Prague, Czech Republic).
- *Model-based design.* (n.d.). In Wikipedia. Retrieved March 10, 2013, from http://en.wikipedia.org/wiki/Model-based_design
- (2012). *ARM Assembly Language Tools.* Texas Instruments.
- (2012). *ARM Optimizing C/C++ Compiler.* Texas Instruments.
- (2013). *Embedded Coder - Reference.* MathWorks.
- (2013). *Embedded Coder - User's Guide.* MathWorks.
- Barry, R. (2009). *Using the FreeRTOS real time kernel - A practical guide.*
- (2013). *Simulink Coder - Reference.* MathWorks.
- (2013). *Simulink - Target Language Compiler.* MathWorks.
- (2013). *Simulink Coder - User's Guide.* MathWorks.
- (2013). *Simulink - Developing S-Functions.* MathWorks.
- (2012). *TMS570LS31x/21x 16/32-Bit RISC Flash Microcontroller - Technical Reference Manual.* Texas Instruments.

9 Appendix A: Notes on FreeRTOS memory management

FreeRTOS provides 4 different (at the time of this writing) memory management implementations. On vanilla distribution of FreeRTOS these can be found in <FreeRTOSRoot>/FreeRTOS/Source/portable/MemMang with the names heap_1.c, heap_2.c, heap_3.c and heap_4.c. The user is supposed to choose one and rename it to heap.c and include it in the port for the target processor. Memory management implementation of each file is explained in depth in:

Memory Management

The above is a must read documentation. In summary:

→ heap_1.c

- Use a static allocated array for memory and thus will be placed on the .bss section.
- Subdivides the array into smaller blocks as RAM is requested.
- Memory cannot be freed.
- Array is as large as configTOTAL_HEAP_SIZE option in FreeRTOSConfig.h.

→ heap_2.c

- Use a static allocated array for memory and thus will be placed on the .bss section.
- Uses a best fit algorithm and allows previously allocated blocks to be freed.
- It does **not** however combine adjacent free blocks into a single large block.
- Array is as large as configTOTAL_HEAP_SIZE option in FreeRTOSConfig.h.

→ heap_3.c

- Wrapper around standard C library malloc() and free().
- Wrapper makes malloc() and free() functions thread safe.
- Memory is as large as defined in linker for C system heap.
- configTOTAL_HEAP_SIZE option in FreeRTOSConfig.h is ignored.

→ heap_4.c

- Use a static allocated array for memory and thus will be placed on the .bss section.
- Uses a first fit algorithm.
- It **does** combine adjacent free memory blocks into a single large block (it does include a coalescence algorithm).
- Array is as large as configTOTAL_HEAP_SIZE option in FreeRTOSConfig.h.

Not all kernels available for the RPP C Library use the same implementation. This is what each kernel is currently configured to use:

Kernel	Origin	Implementation
6.0.4 Posix	Simulator from OpenPilot.org	`heap_3.c`
7.0.2 TMS570	HalCoGen	`heap_1.c`
7.4.0 TMS570	HalCoGen	`heap_4.c`
7.4.2 TMS570	Adapted from vanilla distribution	`heap_4.c`

The relevant implications of this are:

- If a kernel with `heap_3.c` is used the Simulink model *C system heap size* and *Model step task stack size* should be tightly related and the first should be large enough to allocate system tasks and the stack for the stepping task.
- If a kernel with `heap_1.c` is used the programs should not delete tasks, queues or semaphores. If the application spawn and deletes tasks it will eventually deplete the memory available. This is the case with the *rpp-test-suite*. Note that failure to allocated memory from the array will trigger the *Malloc Failed Hook Function* `vApplicationMallocFailedHook()`, even if the implementation doesn't use the C system `malloc()` function.
  Also, *Model step task stack size* should never be set larger than `configTOTAL_HEAP_SIZE` option in `FreeRTOSConfig.h`. Currently the RPP Target doesn't include a GUI option for setting `configTOTAL_HEAP_SIZE` because the library is statically linked and thus memory will be of the size specified when built. The RPP Target **doesn't** check that the requested memory for the step task is less than the `configTOTAL_HEAP_SIZE`, and if greater then the application will fail at runtime and trigger the *Malloc Failed Hook Function*.

10 Appendix B: Known operating-system dependent files

This project was developed on a GNU/Linux operating system. No test for cross-platform interoperability was performed on the code developed. Although care was taken to try to provide platform independent code and tools this are the elements that are know to be Linux dependent:

- LCM1 hardware control tool `lmc1.py`.
  This tool is both GUI and command line capable, the following just affects the GUI part.
  Command line should be usable under Windows systems.

  Cause: Serial port search algorithm is Linux dependent and Gtk 3.0 dynamic Python bindings are not yet available on other operating systems.

- TI CGT support file for RPP Simulink Target `target_tools.mk`.

  Cause: Use UNIX path separator `/`.

- Simulink RPP Target download script `rpp_download.m`.

  Cause: Use UNIX path separator `/`.

- Simulink RPP Target install script `rpp_setup.m`.

  Cause: Use UNIX path separator `/`.

- Simulink RPP Block Library block compilation script `compile_blocks.m`.

  Cause: Call Matlab MEX executable with Unix name.

- All CCS projects under <`repo`>`/rpp/lib/apps/`. Cause: Paths are configure using UNIX path separator `/`.