Costa Rica Institute of Technology

Czech Technical University in Prague

Code generation for automotive rapid prototyping platform using Matlab/Simulink

---

Progress report 2

Author:
Carlos Jenkins

June 25, 2013

Contents

1 Introduction

This document describes the second progress report for the project "Code generation for automotive rapid prototyping platform using Matlab/Simulink" executed on the first semester of 2013 at the Department of Control Engineering, Faculty of Electrical Engineering, Czech Technical University, as part of the specialty practice for applying for the Computer Engineering Bachelor degree at the Costa Rica Institute of Technology.

This report aims to outline the design of the system to develop.

2 Design model

This section describes the architecture of the system to be built. Please note that this project target C embedded development and Simulink code generation (no user graphical interface, no database management system, etc) and thus traditional UML based design methodologies does not apply.

2.1 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 design of this library.

2.1.1 Architecture

The RPP library is 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.

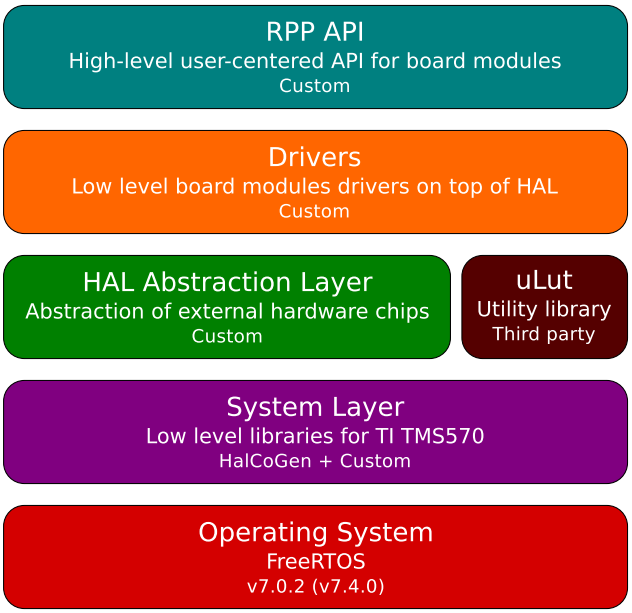

Figure 1: The RPP library layers.

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

2.1.2 RPP Layer Modules

The RPP Layer is 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 will be constantly used on the Software and documentation. Also note that only the following modules will be 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 will be 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.

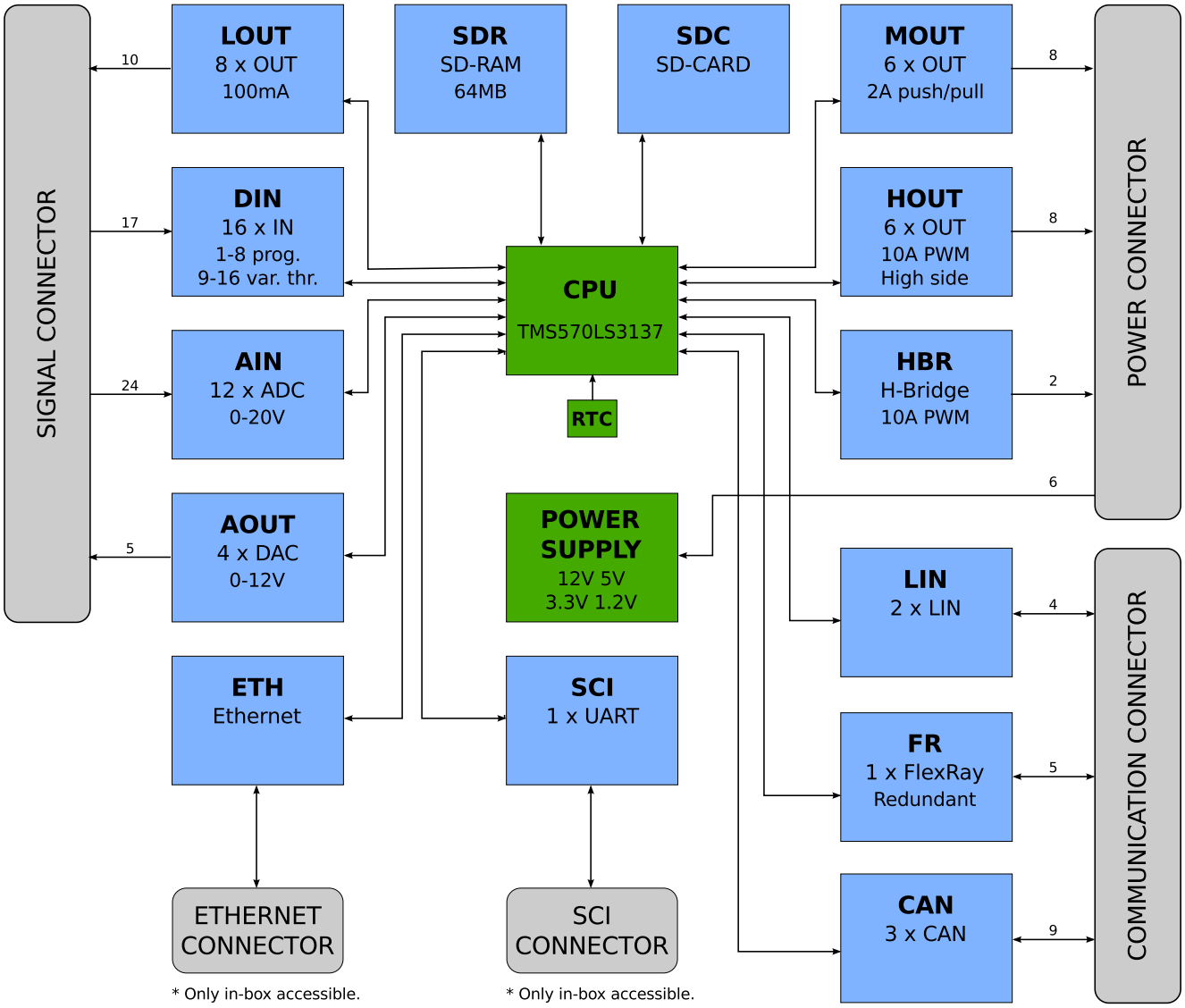

Figure 2: The RPP Library modules.

2.1.3 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.

2.1.4 API development guidelines

The following are the development guidelines that will be used 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.

2.2 Simulink Coder Target

The Simulink Coder Target is the deliverable that will allow Simulink model's code generation, compilation and download for the board.

The Simulink RPP Target will provide 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 will use Texas Instrument ARM compiler (armcl) included in the Code Generation Tools available with Code Composer Studio.

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

2.2.1 Code generation process

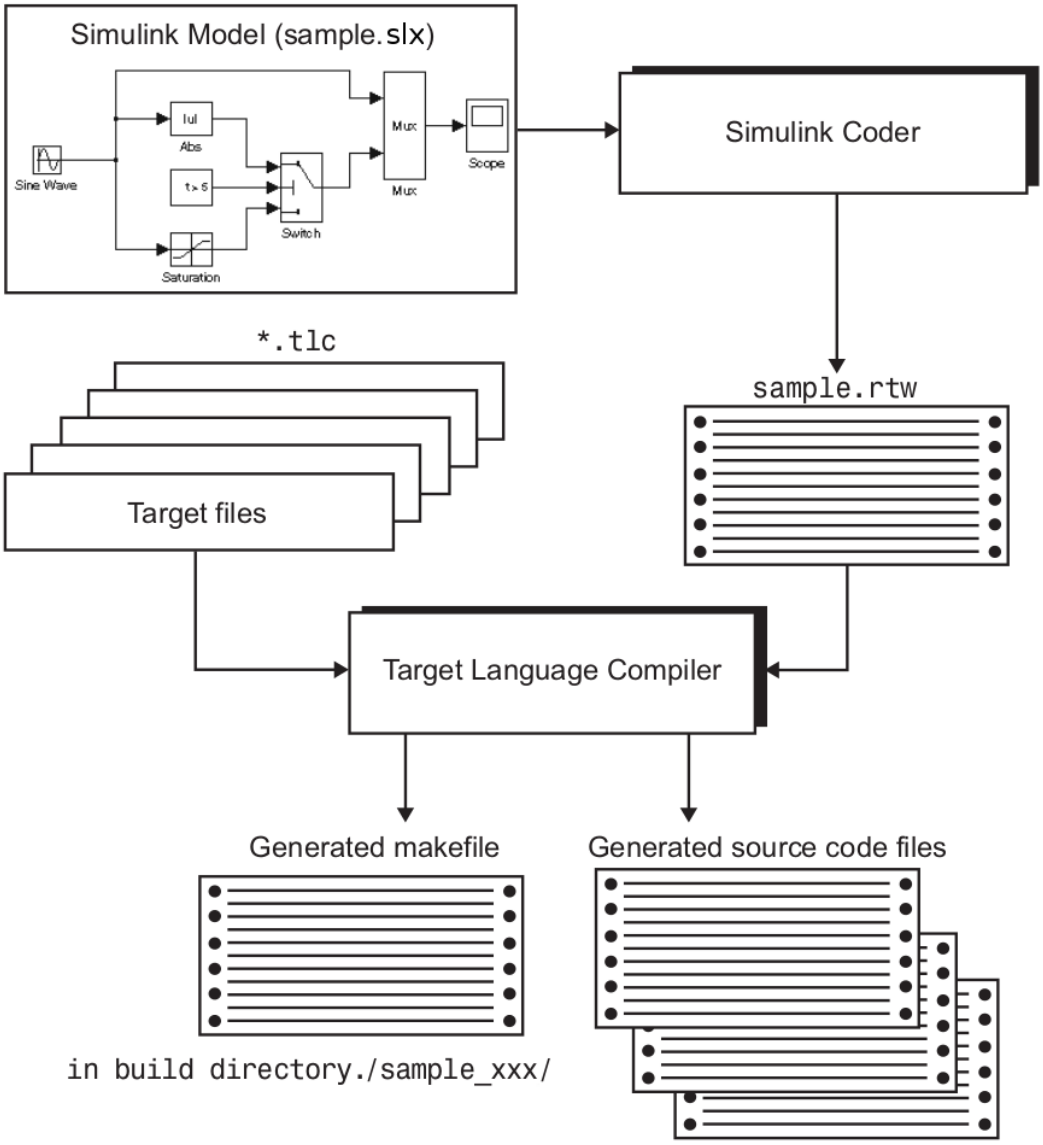

Figure 3: TLC code generation process.

2.3 Simulink Block Library

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

2.3.1 C MEX S-Functions

All of the blocks will be implemented as a C Mex S-Function coded by hand. In the this section the approach that will be 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:

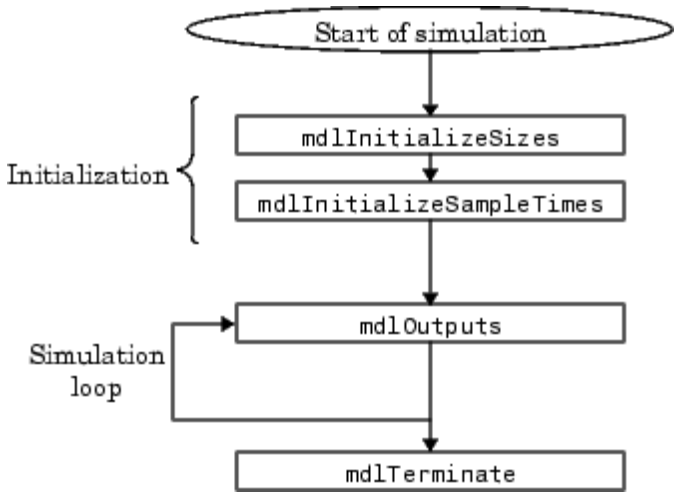

Figure 4: Simulation cycle of a S-Function.

In general, a S-Function can perform calculations and inputs and outputs for simulation. Because the blocks that will be implemented for this project are for hardware peripherals control and IO the blocks will be implemented as pure sink or pure source. That is, the S-Function will be a descriptor of the block but will not do any calculation or 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 will **not** be used 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 hand written S-Functions share 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 will be 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 will reduce 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 will be 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 will 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` will be included that will allows all `sfunctions_*.c` to be fed to the `mex` compiler so all S-Functions are compiled at once.

2.3.2 Target Language Compiler files

C code generated from a Simulink model will be 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 will 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 will be 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 that will be 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 that will be 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.

2.4 Simulink Demos Library

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

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

The following table shows the specification and status of the demos:

Name	**Implemented**	**Tested**
analog_passthrough	NO	NO YET
analog_sinewave	NO	NO YET
digital_passthrough	NO	NO YET
echo_char	NO	NO YET
hbridge_analog_control	NO	NO YET
hbridge_digital_control	NO	NO YET
hbridge_sinewave_control	NO	NO YET
hello_world	NO	NO YET
led_blink_all	NO	NO YET
led_blink	NO	NO YET
log_analog_input	NO	NO YET
power_toggle	NO	NO YET

3 Work schedule

Date	Deliverable	Priority
15.4.2013	C support library and test suite.	1
01.5.2013	Simulink Coder target for RPP board.	2
01.6.2013	Simulink Block library. S-Functions and TLC files.	3
07.6.2013	Simulink Demo library.	4
21.6.2013	Finish documentation of the developed product.	5