Abstract

MOSTER, JOSEPH E. Integration of a Real-Time Operating System into Automatically Generated Embedded Control System Code. (Under the direction of Dr. Charles Hall.)

The emergence of the unmanned aerial systems industry has created an increased interest in the development of autopilots for commercial use as well as research and development. One of the core elements in the development process for an autopilot is the process used to design and implement the flight control software. Typically, the process starts with simulation of the designed controller. Engineers are then presented with the challenge of developing software to run on the flight hardware that accurately executes the designed controller. As further design work is conducted, or as flight data becomes available, it is desirable to improve on the controller design and subsequently propagate those changes to the flight software. Automatic code generation has made this process easier, but lacked the ability to provide for the real-time needs of the controller when using lower power electronics. To address this challenge, a set of software tools was developed to modify the Simulink Real Time Workshop's output to integrate the generated code with a real-time embedded operating system. Testing has validated that code is generated as intended, although the problem of software validation and verification remains an open problem in this work. The software has been successfully shown to generate flight control code from a Simulink block diagram that was then run on an Atmega processor using FreeRTOS to execute the tasks.

Integration of a Real-Time Operating System into Automatically Generated Embedded Control System Code

by
Joseph E. Moster

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Aerospace Engineering

Raleigh, North Carolina

2015

APPROVED BY:

Dr. Larry Silverberg	Dr. Stephen Campbell

Dr. Charles Hall
Chair of Advisory Committee

Acknowledgements

I would like to thank my advisor for his patience and thorough feedback. I would also like to thank my parents. Without their constant encouragement this work may never have been finished.

Table of Contents

List of Figures

Chapter 1

Introduction

Modern unmanned aerial systems have been made possible by advances in microcontroller technology that have led to small chips with ever increasing computational capabilities. A consequence of this increased computational power is that microcontrollers can, and are expected to, handle more tasks concurrently. Handling multiple tasks on a single chip eases inter-process communication and reduces the need for additional processors. This leads to smaller more power efficient boards.

As such, an autopilot may have a variety of jobs in addition to its core task of running the flight controller. These broadly include such tasks as servicing communication requests or supporting payloads. However, there are significant differences between these tasks in terms of both their criticallity to flight and to their real-time requirements. The flight controller is both critical to flight and has tight time tolerences that must be met for it to function properly. Other tasks such as servicing communications are important to the mission, but are tolerant to greater variable delays than the flight controller. With regards to safety, although total loss of communications would disrupt a mission by potentially requiring a return-to-base or other such contingency, it would not result in loss of control of the aircraft. Similarly, tasks such as payload control may be caused to fail completely without affecting the ability of the vehicle to maintain safe flight. To meet the hard-real-time requirements of the flight controller, a real-time multitasking operating system can be used.

Another development has been the ability to automatically generate control system code from a simulation model. By expanding the code generation software to interface with a real-time operating system, an efficient tool is created for the control system developer. This thesis discusses the development of such a tool.

1.1 Code Generation

Simulink is a powerful tool produced by Mathworks for simulating complex systems. Its block diagram interface allows users to quickly prototype control systems and model the response. After a flight control system has been designed and tested, various strategies are used to implement a physical control system that behaves the same way as the modeled system.

To implement the designed control system on flight capable hardware, the traditional design cycle has been to manually code the flight controller. This is typically done in a language such as C that can be compiled to run on the target hardware. There are a few challenges that arise from this process. First, the process is slow. Depending on the complexity of the control system, it can take a significant amount of time to create the final code. The implementation will also vary from author to author, and as such may run slightly differently depending on the exact implementation. The result is that the design cycle is greatly slowed by the coding stage. A maintenance problem is also created since changes to the Simulink model made later must be applied manually.

An improvement on this design cycle is to automatically generate code for the flight controller into embeddable functions. These functions are then manually stitched into an template autopilot that provides the controller with state information such as sensor readings and handles the output commands generated[1]. This speeds up development and reduces time to iterate from days to hours.

The question raised by the semi-automatic process is how much more can the process be automated. It is this fully automatic design cycle that this thesis explores. In this design cycle the process of creating embedded code is completely automatic and transparent to the user. The result is that the transition from Simulink to flight hardware takes seconds and makes iterating easy.

This automation is accomplished using Mathwork's Embedded Coder software which integrates automatic code generation seamlessly into the Simulink environment. The Embedded Coder supports a wide range of processors including x86, ARM, and AVR. Code can also be generated to work with Linux or with Wind River VxWorks[2]. However, while VxWorks is an embedded Real-Time Operating System, it cannot be run on the low-power hardware used in this project[3].

1.2 Real Time Operating System

For a control system to operate correctly there are temporal requirements that it must meet. Sensors must be read and outputs must be generated at a specified rate otherwise the control system may not function as designed. The primitive method to accomplish this is to write the

code so that all intermediate functions return fast enough that the operation of the critical control system functions is not disrupted. This method does not guarantee that timings will be met and can lead to excess complexity and poor performance. A real-time operating system (RTOS) addresses these issues.

An RTOS goes beyond a typical operating system by adding priority level as a factor in the task scheduler. This inclusion allows for higher priority tasks to interrupt lower priority tasks. In this way, a control loop task can pause the execution of a low priority task, such as telemetry messages, so that it can meet its time requirement. This allows for time consuming low priority support tasks to be included without endangering the performance of the control system.

Chapter 2

Background

This chapter explores the inner workings of the RTOS and Simulink code generation. Specifically, this includes an introduction to how the RTOS handles tasks, and how Simulink generates code from its block diagrams.

Throughout this thesis, the test hardware used was an Ardupilot Mega V1.0 with a sensor shield. This commercial-off-the-shelf board is cheap, small, and has minimal power requirements. The board uses the ATmega1280 chip by Atmel which, importantly, includes four 16-bit timers that can be tied to hardware interrupts[4]. This allows an RTOS to run on the hardware. Additionally, Simulink supports code generation for the ATmega1280 and has a blockset supporting several basic functions. The "Embedded Coder Support Package for Arduino"was used with the Embedded Coder as the starting point for development.

2.1 FreeRTOS Real-Time Operating System

There are a number of RTOS on the market targeted at embedded applications. For this work, FreeRTOS was chosen for its compatibility with the test hardware and its maturity as a project. FreeRTOS is a preemptive multitasking operating system capable of meeting hard-real-time task requirements [5]. Since FreeRTOS is compatible with a variety of processors it was necessary to choose an appropriate port of the project that would work with the test hardware. The DuinOS port of FreeRTOS is compatible with the AVR architecture used by the ATmega series of chips, enabling FreeRTOS to run on the ATmega1280 chip among others [6].

Within a RTOS the functions that the OS is to run are organised into tasks. In a multitasking OS, such as FreeRTOS, each of these tasks operates concurrently with the other tasks running on the system. Each task is given a priority level and a stack size. By default, three priority levels are available, namely high, normal and low priority. Additional priority levels can be added if needed. Since the stack size can be set for each individual task, the default stack size

of 85 bytes may be used, or for tasks needing a larger stack, the stack size can be increased.

Declaration of a task is accomplished in three steps. First a forward declaration of the task's function is made. Then the task function is declared. This is done using a macro which wraps the function's content in an infinite loop [7]. Now the task may be added to the task scheduler. This is typically done in the program's initialization loop.

2.2 Simulink Automatic Code Generation

Although code generation is a multi-step process involving a large number of intermediate files, the process is completely automatic and transparent to the user [8]. After the user clicks the build button, no further interaction is needed before the firmware is burned to the microcontroller. The process starts with the user's Simulink model which is converted to the intermediate Real Time Workshop (RTW) document by SFunctions. The RTW document is then converted to C using Target Language Compiler (TLC) files. In Figure 2.1 the process and associated file types are enumerated.

Figure 2.1: Overview of code generation process

In the first step, the simulink blocks are converted into the RTW format which completely describes each block including its type, parameters, and connections to other blocks. The final RTW document created by this process also contains global model information that includs settings and statistics of the overall model. The conversion of individual blocks is accomplished with S-Functions associated with each type of block. This includes user defined blocks whose conversion is handled by S-Functions that are written for each custom block.

S-Functions are software modules that are used to extend the capabilities of Simulink. By using the S-Function API, these modules can be loaded by Simulink and communicate with the Simulink engine. The S-Functions created for this thesis are written in C and compiled into a *.mexw32 or *.mexw64 file that Simulink can use (listed as .mex in Figure 2.1) . Using the aformentioned S-Function API, the S-Function takes information from the Simulink block, such as its user defined parameters and sample time, and adds that information to the RTW

document. The S-Function can also be used to check for bad user inputs.

The complete RTW document is used by the Target Language Compiler to generate the embedded code. This embedded code can be divided into three groups: entry point, tasks, and auxiliary. The flowchart in Figure 2.2 shows the transition from the RTW document to embedded C. The RTW file is shown in purple, while the TLC files are shown in Red and the generated code is shown in orange. In this example the control system "myCtrl" has been converted to an RTW document.

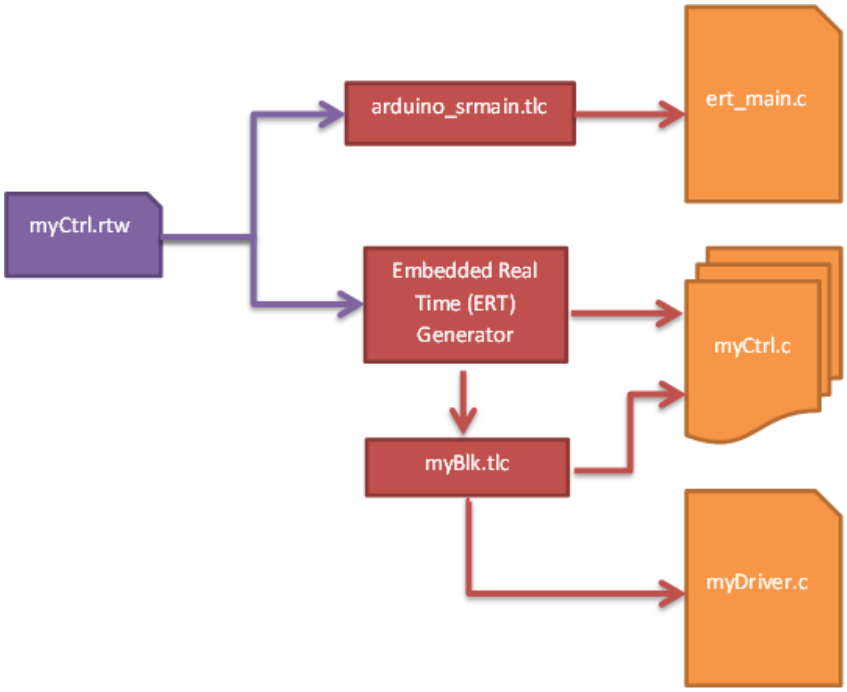

Figure 2.2: Code generation from Real-Time Workshop document

The entry point of the firmware is generated by a special TLC file specified as the "System Target File" (STF). Its purpose is to instruct the Simulink Coder how to generate code for particular environment [9]. For embedded applications, it is designed to setup and then execute the code at boot.

The tasks themselves are created by the Embedded Real Time (ERT) generator, a specific set of TLC files for embedded code. This generator calls TLC files corresponding to each block described in the RTW document to generate its specific lines of code. The end result is a set of files that can be compiled into the final executable. The key files that contain the task's

contents have the same name as the initial Simulink model, but with a different suffix and file extension. The system model C file created contains all of the task functions as well as a function to initialize the code. A full description of all the files created by the ERT generator is available in Mathwork's documentation [10].

Some TLC files may also generate entirely independent C files, such as custom device drivers, that bring specific functionality to the code. In Figure 2.2 the TLC file "myBlk.tlc" generates its own C file in addition to adding its own code into the main application.

2.2.1 Task Generation

Within the Simulink graphical environment, there isn't normally a clear sense of tasks. To create a semblance of tasks, it is instructive to look at when a particular block is called. From this perspective there are three ways a block can be called for execution: once at initialization, by a function-call, or periodically based on its sample time.

The first type of blocks are called once at initialization. These are typically configuration blocks that are needed for setup, but are not regularly run afterwards. An example of this kind of block is the "ADC Config" block in Figure 2.3. This block is responsible for setting up pins to function as analog to digital inputs. Another type of block run at initialization are blocks that have a sample time of "inf". Since the output of these blocks never change, they can be initialized once at start-up.

The next way that blocks may be called is by a Function-Call block. The Function-Call block is placed in the top level diagram and is connected to a Function-Call Subsystem block. The Function-Call Subsystem block internally contains a block diagram that is executed when commanded to by the Functional-Call block. The Function-Call block itself determines when it should call the blocks connected to it. As the function-call blocks are explicitly defined by the user they are referred to as explicit tasks in this paper. The function-call blocks are shown in green in Figure 2.3.

Finally, blocks outside of a Function-Call Subsystem are called based on the sample time of the block. These blocks are given a sample time by the user or inherit a sample time that determines how often they should run. Blocks may share the same sample time even if they are not connected or correlated to each other. Since these blocks are grouped without regard to their function and without input from the user they are referred to in this paper as implicit tasks.

An example of blocks that feed into implicit tasks are the "Rate Transition" blocks in Figure 2.3. Rate transition blocks are placed outside of tasks to handle the passing of data from one task to another task running at a different rate. For example, the high-speed State-Estimator task passes state information to the medium-speed Controller task via the block

named “Rate Trans”.

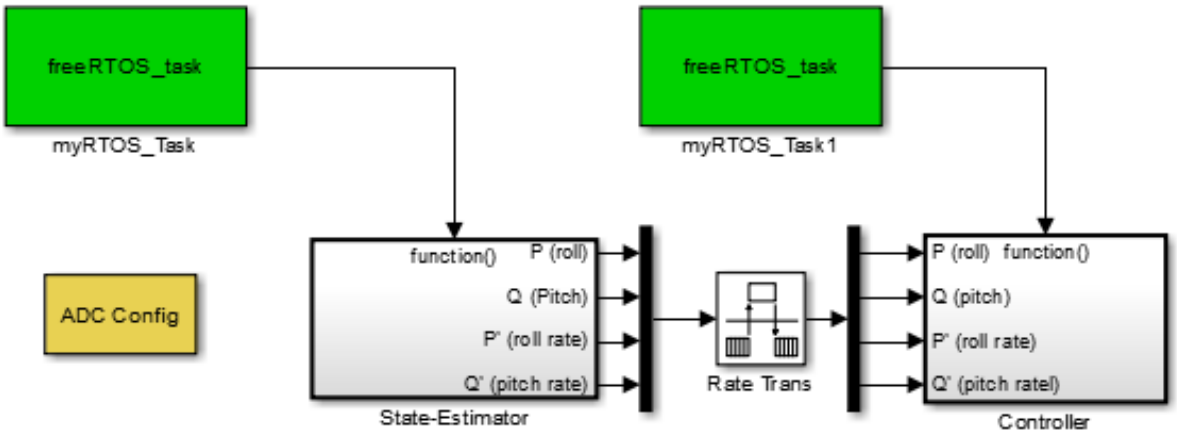

Figure 2.3: Diagram illustrating the various types of blocks

Chapter 3

Implementation

Integration of the RTOS into Simulink started at the highest level of Simulink, the diagrams. This was done to make working with the RTOS simple and intuitive for the user by presenting the new functionality in the same way as regular Simulink blocks. The standard way of adding new capabilities to Simulink is to add a new blockset which acts as a library for a collection of Simulink blocks. As such, a blockset was created to expose the functionality of the RTOS as Simulink blocks. The parameters of these blocks could then be manipulated through their "masks" which are custom defined graphical user interfaces (GUI). The blockset also included S-Functions and TLC files to integrate the information from the new blocks into the automatically generated code.

The new "Free RTOS" blockset contains two blocks, the freeRTOS_task block for setting up explicit tasks and the NICP block for inter-task communication. The freeRTOS_task block was needed so that the user defined properties of a task including name, stack size, and priority, could be linked to a particular sub-diagram of blocks that define the functionality of the task. The NICP (Non-Interruptible Copy) block provides a mechanism for safely transferring data between tasks. Without a mechanism such as this, data copying could be interrupted by a context switch resulting in corruption of the information.

Additionally, modifications were made to the TLC files that were a part of the base code generation capabilities to use the new blocks' information in generating the structure of the code.

3.1 Simulink Blocks

The freeRTOS_task block was created as a special type of Function-Call block to define the properties of explicitly defined tasks. The user connects it to a Function-Call Subsystem block that contains the blocks describing its task. An example top-level block diagram of a model

setup to generate real-time code is shown in Figure 3.1.

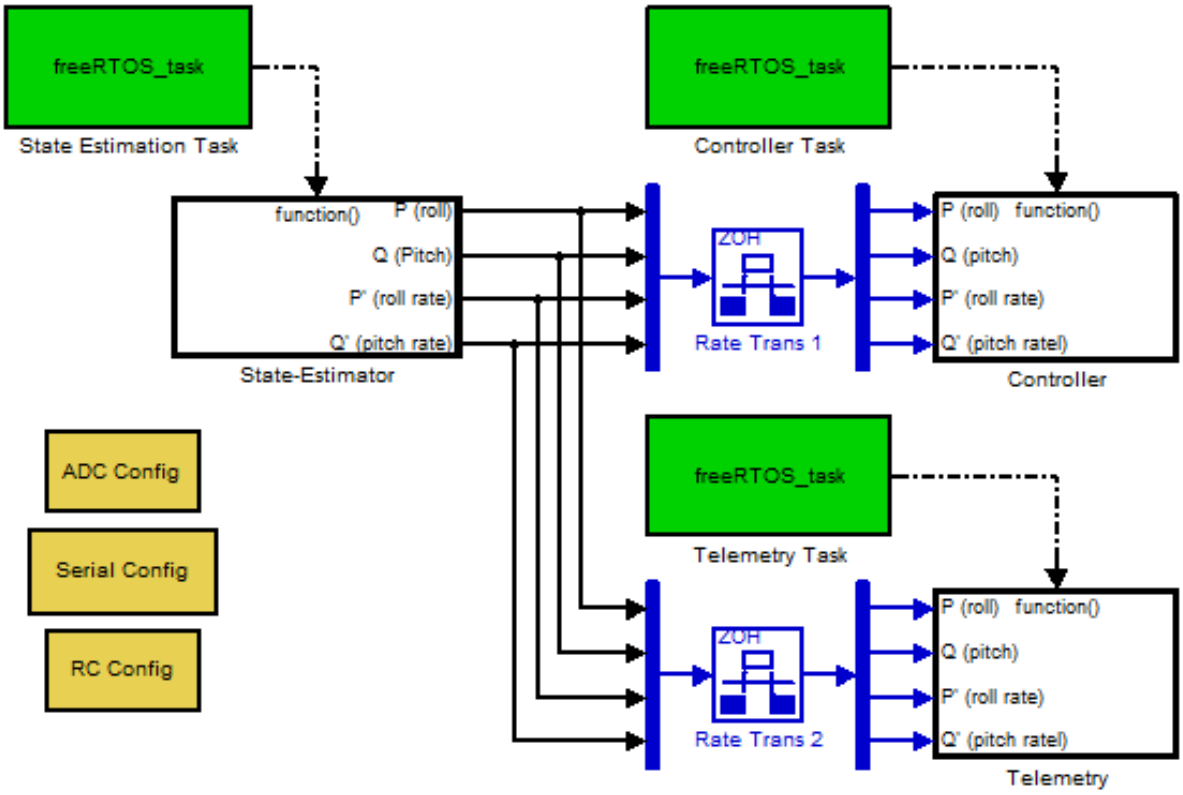

Figure 3.1: Top Level diagram showing a system configured for RTOS integration

In this block diagram, freeRTOS_task blocks are connected to Function-Call Subsystem blocks labeled "State-Estimator", "Controller", and "Telemetry". Each of the Function-Call Subsystem blocks contain a block diagram that describes the task that the subsystem performs. Each of these block diagrams operates at a single rate and has input and output blocks as needed to communicate with other tasks.

To improve usability, a block mask was implemented for the freeRTOS_task block. By double-clicking on the freeRTOS_task block the user can see its options as shown in Figure 3.2. The information entered into this interface will be used by the block's S-Function when adding the task information to the RTW file. This allows the user to easily set the task's name, priority, and stack size. By limiting and simplifying the possible range of inputs, common errors are eliminated. For example, the block mask will only allow the user to select a valid task priority

level. Additionally, malformed inputs can be detected and flagged as soon as the user tries to apply the erroneous change. For example, the stack size field only allows positive integer values to be entered. Values that include letters would be rejected.

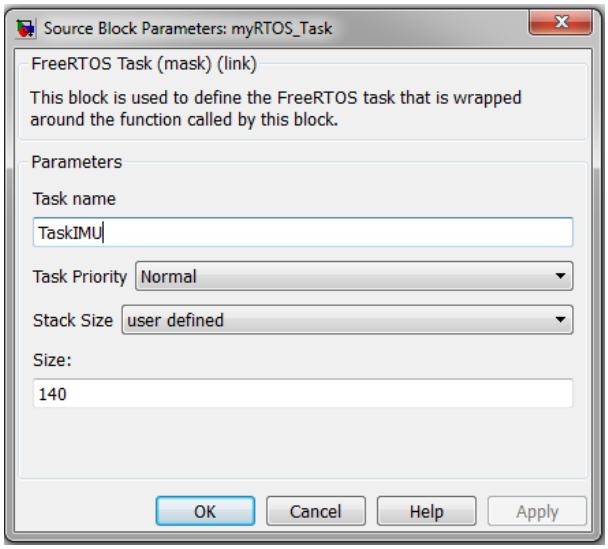

Figure 3.2: Explicit task block mask

Another important feature was for tasks to be able to communicate with each other. This was accomplished using the standard Rate-Transition block and the NICP block.

The NICP block was placed within a task's subsystem block immediately before any output blocks and copies data that it receives into an output variable. Since it does so in a way that cannot be interrupted, it is guaranteed that any other task accessing the output variable will never see a value that has been partially written to. Because the NICP block does not have any parameters it did not need a block mask like the freeROTS_task Block.

As shown in Figure 3.1, inter-task communications are connected through Rate-Transition blocks which are located outside of any one task. This block is a standard part of Simulink and was not subsequently modified. When triggered, the Rate-Transition block copies data from the NICP's output variable to its own internal copy. Since blocks in top level diagram are run at the highest priority this copy transaction cannot be interrupted. This prevents the output of the function from being corrupted by an interrupt. The Rate-Transition blocks also satisfy a Simulink requirement to explicitly determine how block diagrams running at different rates exchange data.

Finally, a second NICP block connected to the input block of the receiving task ensures that the data being read will not be altered until it has finished making an internal copy of the data.

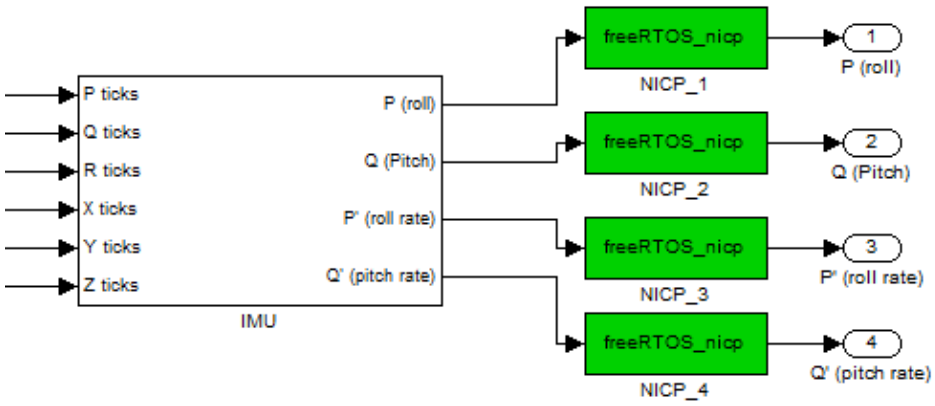

Figure 3.3: Contents of a Function-Call Subsystem showing Non-Interruptible Copy blocks

Beyond the FreeRTOS blockset, other blocks were created for the purposes of testing or demonstrating functionality of the FreeRTOS blockset. These included blocks for configurating, reading, and writing to various hardware peripherals. Some examples include outputting motor commands and reading data from the inertial measurement unit and RC receiver. Some of these blocks for setting up peripherals during initialization are shown in Figure 3.1 with orange backgrounds.

3.2 S-Functions

For each Simulink block created there was also a corresponding S-Function that needed to be made as well. At a minimum, each S-Function needs to define several callback methods to fulfill its requirements as an S-Function. An S-Function may also contain functionality to validate parameters that it has been given. Since all the data gathered by the Simulink block is passed to the S-Function, the S-Function can check the validity of the received parameters before adding them to the RTW document. If invalid parameters are detected, appropriate errors can

be thrown to describe the issue to the user. Since the NICP block does not have any parameters that it needs to validate, it only needs to define the required callback methods[11].

The S-Function for the freeRTOS_task block required much more functionality. The validation for each of the freeRTOS_task S-Function's three parameters is distinct. The priority level parameter is the easiest to validate as it must be one of only three supported values. The stack size parameter is also simple as it is only verified as being an integer greater than or equal to the minimum size. It is important to note here that this thesis work does not handle memory allocation issues that can arise if too much stack space is allocated across the various stacks.

The more complicated parameter to check is the task name parameter. The task name parameter will become part of the function name identifier. This function name identifier is validated to be in compliance with the identifier rules as described in section 6.4.2 of the C99 standard ISO/IEC 9899:1999 [12]. As it is currently written, the validation is more strict than necessary because it currently does not allow underscores in the name.

3.3 TLC Files

In the final stage of code generation, the RTW file now containing the special Simulink blocks' data added by the S-Functions, is converted into executable code. To add support for a real time operating system it was necessary to rewrite the entry point of the code as well as provide support for the generation of implicitly and explicitly defined tasks. This required the creation of three TLC files, "freeRTOS_task.tlc", "freeRTOS_nicp.tlc", and "arduino_srmain.tlc". The later of which uses the same name as the stock tlc file used by Mathworks to generate non-OS code. A line-by-line examination of "freeRTOS_task.tlc" and "arduino_srmain.tlc" is available in Appendix D.

The entry point of the code, the function "main()", is created by the arduino_srmain.tlc file. This function is responsible for setting up the environment, tasks, and starting the scheduler. Figure 3.4 shows the main function generated for an example Simulink model "IMU_tasked". As shown in the example, the main function first calls the "init()" function to setup the basic hardware excluding any peripherals[13].

After that, the initialization function for the model is executed; the function is named "IMU_Tasked_initialize" in this case. This function contains model specific code from a number of blocks including the freeRTOS_task block and is located in the "IMU_Tasked.c" file. The function allocates memory for the various structures needed and adds the explicit tasks that have been defined in the IMU_Tasked.c file. It finishes by calling any block's functions that need to be run at initialization. This includes blocks such as those with orange backgrounds in Figure 3.1 that set up hardware peripherals.

Once the model initialization function returns, the main function then adds the implicit tasks

```c
//IMU_Tasked.c
//For each task
declareTaskLoop(TaskIMU_fcn);
taskLoop(TaskIMU_fcn)
{
  unsigned long previous = millis();
  {
        //Task functionality goes here
  }

  if (0.025*1000>(millis()-previous)) {
    delay(0.025*1000-(millis()-previous));
  }
}

//For each task
declareTaskLoop(Task_fcn);
taskLoop(Task_fcn)
{
  unsigned long previous = millis();
  {
        //Task functionality goes here
  }

  if (0.05*1000>(millis()-previous)) {
    delay(0.05*1000-(millis()-previous));
  }
}

void IMU_Tasked_step0(void)            /* Sample time: [0.025s, 0.0s] */
{
  //Rate Transition code goes here
}

void IMU_Tasked_step1(void)            /* Sample time: [0.05s, 0.0s] */
{
  /* (no output/update code required) */
}

void IMU_Tasked_initialize(void)
{
  //Memory allocation goes here

  createTaskLoopWithStackSize(TaskIMU_fcn, NORMAL_PRIORITY,140.0);

  createTaskLoopWithStackSize(Task_fcn, LOW_PRIORITY,85.0);

  //Initialization code for peripherals goes here
}
```

```c
//ert_main.c
//For each task
declareTaskLoop(stepTask0);
taskLoop(stepTask0)
{
  unsigned long previous = millis();
  IMU_Tasked_step0();

  if (0.025*1000>(millis()-previous)) {
    delay(0.025*1000-(millis()-previous));
  }
}

declareTaskLoop(stepTask1);
taskLoop(stepTask1)
{
  unsigned long previous = millis();
  IMU_Tasked_step1();

  if (0.025*1000>(millis()-previous)) {
    delay(0.05*1000-(millis()-previous));
  }
}

int_T main(void)
{
  /* Initialize model */
  init();
  IMU_Tasked_initialize();
  createTaskLoop(stepTask0, HIGH_PRIORITY);
  createTaskLoop(stepTask1, HIGH_PRIORITY);

  //Task Scheduler
  vTaskStartScheduler();

  //Will not get here unless a task calls vTaskEndScheduler():
  for (;;) ;

  //    IMU_Tasked_terminate();
  return 0;
}
```

Figure 3.4: Generated code example showing IMU_Tasked.c on the left and ert_main.c on the right

defined in the ert_main.c file. After this, the main function finishes by starting the scheduler. This function will not return unless the scheduler is stopped by a task; a capability that is not currently made accessible to the user.

Outside of the entry point code, tasks must be declared and then added to the scheduler. The declaration sets up the tasks loop, its timing, and establishes what code shall be run. Implicit and explicit tasks are set up differently because of the scope of information available at different steps in the generation process. However, it is notable that the contents of both implicit and explicit tasks are contained within the model's C file. As such, the differences between the code structure of the two types of tasks is limited to where the tasks are declared and then added to the scheduler.

Implicit tasks are declared in the ert_main.c file by the arduino_srmain.tlc. This is the only TLC file that is given access by Simulink to an array containing information on each implicit task. This information includes the number of tasks which is used to generate the task declarations for all the implicit tasks. For implicit tasks, the task declaration calls a corresponding function in the model's C file that executes the functionality of the task.

In contrast, information on each explicit task is contained within its own section of the RTW document corresponding to the freeRTOS block that defined it. Each instance of the freeRTOS block's information is processed by the freeRTOS_task.tlc file which creates the task definition. Since these blocks are a part of the model definition, the resulting code is all contained within the model's C file. The task declaration is done immediately before the declaration of the task's function, and the addition of the task to the scheduler is completed in the initialization function.

The final TLC file to discuss is the freeRTOS_nicp.tlc file. Unlike the other TLC files discussed here, this file does not modify the structure of the code. It is simply integrated as a part of the tasks themselves. It supports safe copying of data between tasks by copying output values into separate variables that will be read by subsequent accessors. Because it declares the copying code as "critical code", the copying can't be pre-empted by another task. This guarantees that the values read by the Rate-Transition block are not part-way through being written to. Without this protection it would be possible that an interrupt could occur between bytes of a multi-byte variable being written to memory. If the task that is now being executed then tried to read the partially written to value, the value could be wrong. Since this error is dependent on the coincidental alignment of an interrupt with the memory write, it can occur infrequently and randomly making it difficult to debug.

Finally, the file "arduino_make_rtw_hook.m" was modified to allow multiple tasks to be generated. This was accomplished by removing the error thrown by the code in the section marked "No support for multi tasking mode".

Chapter 4

Testing and Evaluation

After software development was completed, testing was conducted to verify the proper functioning of the generated code and the user interface. In particular, testing focused on context switching, clock stability, stack sizing, and input sanitation. What follows are the details of the tests and analysis completed to address each of these.

4.1 Context Switching

Context switching is a basic functionality that is central to the proper operation of any multitasking operating system. The purpose of this test was to show that a higher priority task can interrupt the execution of a low priority task and resume its own execution. Once the higher priority task is finished, the execution of the lower priority task should be resumed. This test also served to demonstrate that the tasks are being executed at the proper rate.

To generate the code for this experiment, a block diagram was created with a pair of tasks configured to each toggle an LED. The LEDs were part of the target's hardware and were choosen for their ease of access and visual confirmation. LED A was connected to pin 37 of the processor while LED B was connected to pin 36. The first task was given high priority and set to execute every 10 milliseconds. This high priority task toggles LED B each time it executes. The second task was set at low priority and set to execute every 20 milliseconds. This task starts by setting LED A high. It was designed to then use up resources by continuously checking the clock until 16 milliseconds have passed. Once this requisite time has passed, it then sets LED A low. This is similar in concept to a spin-lock[14]. However, instead of waiting for a resource lock to be freed it is waiting for a time interval to pass. As such, it is guaranteed that for the high task to toggle its pin every 10 milliseconds it will have to run while the low priority task is still checking the clock.

To observe this happening, an oscilloscope was connected to record the state of the output

pins driving the LEDs. The data from the oscilloscope in Figure 4.1 shows the high priority pin's state on channel 1 while channel 2 shows the low priority pin's state data. The data shows that the high priority state was able to update while the lower priority task was still running. This result is only possible by the higher priority task interrupting the execution of the lower priority task before subsequently restoring the context of the lower priority task. This shows that context switching if functional such that both tasks were able to execute as effectively as they would have had they been running alone.

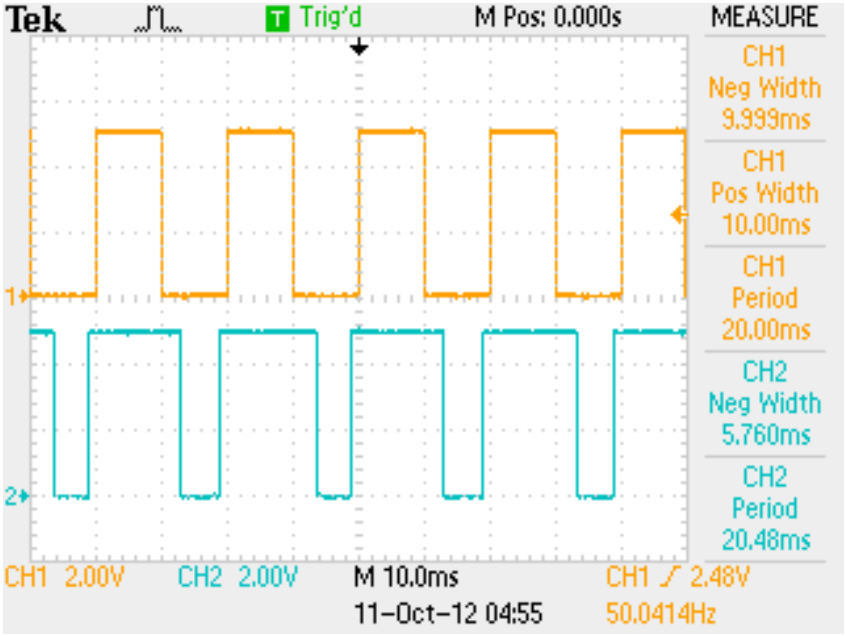

Figure 4.1: Oscilloscope data showing states of the pins

4.2 Clock Stability

The control system being run by the RTOS is dependent on the operating system to execute it at the commanded rate. This requires that the RTOS have an accurate source for timing information. The RTOS came configured to get its timing from an interrupt which is triggered by the 16-Bit Timer 1. The timer is configured to receive "ticks" generated by the external oscillator installed on the test hardware. These ticks first go through a "prescaler" which generates 1 tick for every 64 ticks received. This prescaler prevents the counter in the next stage from overflowing

before it has counted enough ticks to reach the desired time interval. The counter driven by the ticks from the prescaler triggers the interrupt when a comparator detects that the count has reached a specified value[15]. The specified value is determined by the equation (SOURCE_-HZ/RTOS_HZ)/PRESCALER-1. The RTOS frequency was set to 1000Hz to give a 1ms tick rate with the source running at 16Mhz and the prescaler set to 64.

The following experiment was conducted to characterise the performance of the timing of the interrupt triggering. To time the triggering of the interrupt, code was added to the FreeRTOS function vPortYieldFromTick(). This function is a part of the port.c file in FreeRTOS which defines the chip specific code for handling context switching and interrupts. This function is called by the Interrupt Service Routine (ISR) "TIMER1_COMPA_vect" that is being triggered every 1 ms. The modified vPortYieldFromTick function, shown below, toggles the variable "state" and then writes the result out to pin 35 (LED C). When run, pin 35 will toggle each time the ISR is triggered making it observable using an oscilloscope. The state variable itself is initialized at the top of the port.c file. Pin 35 was set to be a digital output by adding the necessary code to the main() function in arduino srmain.tlc. Since the control system under test does not affect the functioning of the timing mechanism or the vPortYieldFromTick function, the block diagram from the context switching test was used.

```
void vPortYieldFromTick( void ) __attribute__ ( ( naked ) );
void vPortYieldFromTick( void )
{
        portSAVE_CONTEXT();
        vTaskIncrementTick();

    //Interrupt timing code start
    state = !state;
    digitalWrite(35,state);
    //Interrupt timing code end

        vTaskSwitchContext();
        portRESTORE_CONTEXT();

        asm volatile ( "ret" );
}
```

Initially the oscilloscope was configured to trigger for two cases of pulse lengths; "equal to" and "not equal to". The oscilloscope considered the pulse length equal to 1ms if it was less than 5% from 1ms. This 5% tolerance is manufacturer defined for the TDS2022B oscilloscope used and could not be altered[16]. The oscilloscope was calibrated prior to the experiment using the built-in calibration source.

During the experiment, no pulses were registered in the "not equal to" case. In the "equal to" case the average frequency was 500.42 Hz. This indicated that overall the signal was 0.1% off from the expected 500 Hz signal. The type of crystal used on the APM hardware has a

frequency tolerance of ±0.5% according to the crystal's datasheet[17]. This puts the overall error of the interrupt timer within the frequency tolerance of the crystal.

In a second experiment, the oscilloscope was used to measure the time between a pair of interrupt triggers which yielded Figure 4.2 which shows a histogram of 80 readings. Each bin is 10 microseconds wide and is labeled by its middle value. The data shows that the average pulse length was 1001 microseconds with a 12 microsecond standard deviation. This translates to a 499.5±6 Hz signal which is 0.1% off from the ideal 500 Hz frequency.

Although the two experimental methods indicated errors in different directions from the ideal signal, the absolute difference is small. The data gathered in these two tests shows that overall, the interrupt is being triggered with an acceptable level of accuracy even though the time between any two consecutive triggers may vary. Differences in the variance could be related to small changes in temperature which can effect the frequency output of the oscillator.

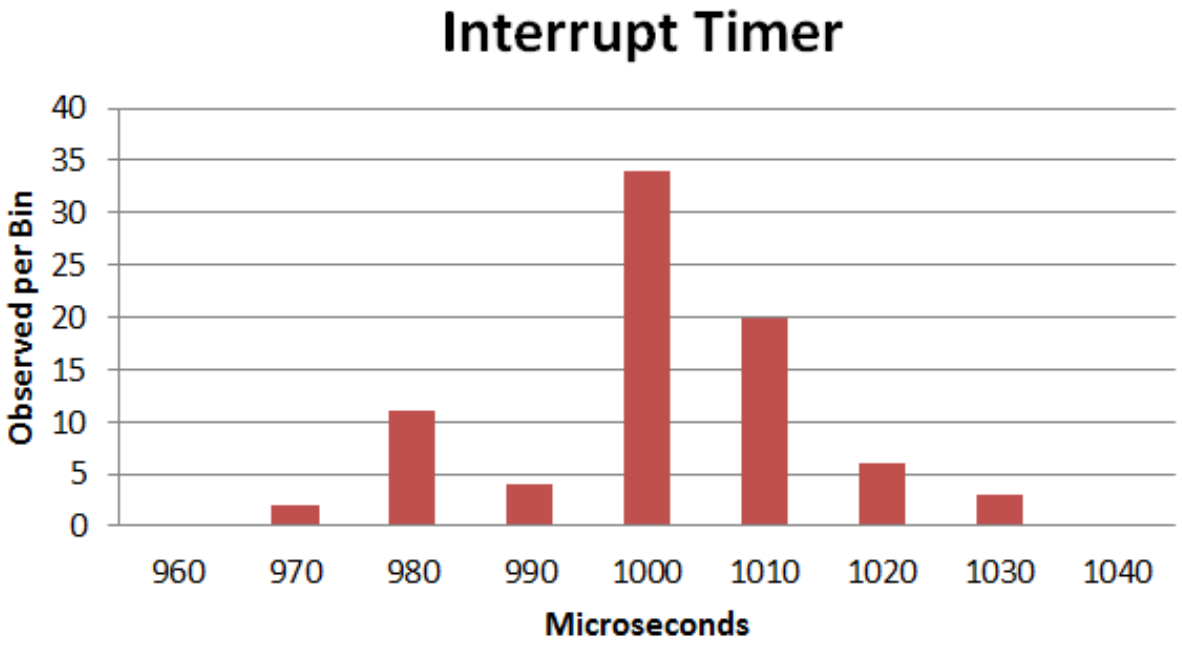

Figure 4.2: Histogram of time between interrupt triggers

4.3 Stack Size

A task with a large stack size, or the potential to grow a large stack, poses a challenge to the operating system which must allocate enough space for the stack or risk a stack overflow condition. Although determining how much stack space to allocate is a challenge beyond the scope of this work, it is important to demonstrate that if commanded to supply a larger stack size, the change will be incorporated into the generated code. In this experiment, it is shown that setting an increased stack size allocation is effective in allowing a task to successfully run that would otherwise overrunning its stack.

First, a task that will overrun its stack was created. The task reads each of six ADC channels and wrote their respective values to the serial port. The task's stack size was set to a size of 40 bytes. A quick look at the generated code revealed that a large number of temporary variables were being instantiated. When the program was subsequently run on the microcontroller, the program crashed immediately.

To determine that there was a stack overflow issue, the stack overflow detection capability of FreeRTOS was enabled. By enabling the configCHECK_FOR_STACK_OVERFLOW parameter in the FreeRTOSConfig.h file, FreeRTOS was instructed to test for stack overflow conditions. This is accomplished by checking if the bottom of the stack that the OS has switched away from exceeds the allocation for that stack[18]. It is notable that in the event that the bottom of the stack overflows its allocation but subsequently recedes to within its allocation before any context switch occurs, the stack overflow will not be detected. However, while this method will not catch all stack overflows, it is sufficient for this experiment to show that there has been a stack overflow.

When a stack overflow is detected, the function vApplicationStackOverflowHook is called. This function was then defined, as shown below, to set pin 35 (LED C) high. The hook function was written at the end of the tasks.c file belonging to freeRTOS.

```
#if (configCHECK_FOR_STACK_OVERFLOW > 0)
void vApplicationStackOverflowHook( xTaskHandle *pxTask, signed char *pcTaskName )
{
   //LED C = Pin 35
   pinMode(35,OUTPUT);
   digitalWrite(35, 1);
}
#endif
```

Now, when the program was run with a stack size of 40, the program stopped within 1 second and LED C came on to indicate a stack overflow. The stack size was then increased to 140 using the "User Defined" stack size option in the task block's mask. After the stack was adjusted, the program was able to run without error and without triggering the stack overflow hook. This shows that the increased stack size was now sufficient to run the task, although from

just this information it is uncertain how much extra stack space was available. However, while this doesn't ensure that a stack overflow cannot happen, it does show that given a desired stack size the generated code will incorporate that change.

4.4 Input Sanitation

Verification of the final system was a significant challenge. A full verification process could include verifying the block diagram, verifying static generated code, and dynamic analysis of code. However, for this work the focus was on verifying that the code generated by the custom blocks will be generated as designed. To this end, verification focused on unit tests designed to capture the full range of possible user defined parameters to the S-Fuctions. Since the NICP block did not have any parameters, it was only the three parameters of the freeRTOS_task block that required validation. Successful input sanitation meant not only catching bad inputs, but also providing usable feedback on the error to the user.

The implementation of input sanitation is discussed in Section 3.2. Should any of the input checks fail, the desired result is an appropriate error message be presented to the user. To aid in testing this functionality, additional code was written to automatically modify block parameters, trigger automatic code generation, and capture any errors and their associated message. The error could then be compared against the expected error for each type of invalid block parameter. The test results were written to an html file that lists the error message and whether the result matched the expected result will a PASS/FAIL field for each result.

The test results were broken down into three sections with each corresponding with one of the three parameters. Since each of the checks is done independently of the other parameters it was not necessary to test combinations of bad inputs. The net result of these tests was complete branch coverage of the code. In total, 19 unit tests were run against the FreeRTOStask block. The scripts used to conduct the testing and the final results can be found in Appendix B. The current test results show that the software passed all the tests and provided the correct error messages.

Chapter 5

Results and Conclusion

The tools presented here have been successful in bringing the advantages of a RTOS to automatically generated code. Now, a typical user can create prioritized multi-tasked code using only the simulink interface. This has opened up the these real-time capabilities to a new class of users including students and controls engineers who would otherwise lack the skills to implement such a system on their own. A user employing this work may focus attention on the capabilities of the flight controller instead of on the mechanisms for actualizing it in hardware. Additionally, the speed and ease of generation allows for quicker iteration and testing of systems. This also creates future opportunities for easier certification of flight controllers by introducing a common tool and framework. Confidence in the system build by one user's testing may be translated to other user's flight controllers as a result of their underlying similarities that result from using a common tool.

The testing conducted has validated the correct functioning of core capabilities. Among these, context switching has been shown to be functional, thus enabling FreeRTOS to interrupt lower priority tasks to run higher priority tasks. The clock is stable enough that FreeRTOS can use it to time when to run the tasks. The stack size settings correctly adjust the size of the stack allocation allowing for large tasks to be run. Finally, unit testing has shown that inputs are correctly sanitized preventing unanticipated code from being generated.

Taking a broader view shows that there remain some challenges. These challenges represent areas where future work could expand the capabilities of this project, but also areas of weakness in the generated code.

First, there are a limited number of priorities. This may pose a challenge to large control systems needing fine control of prioritization. While the current code only supports the three default priority levels that came with FreeRTOS it would not be a significant problem to add additional priority levels.

Currently, setting the stack size is accomplished through trial and error. Other techniques

could be used to improve this process including static analysis and dynamic analysis tools. These methods would have the advantage of automating the process as well as having greater confidence that the stack size will be sufficient. However, with or without these tools, it is still advisable to eliminate potentially unbounded recursion from the control system due to the limited amount of memory available.

For each sample time in the model, Simulink creates a corresponding implicit task. This may result in the creation of implicit tasks that contain no code if a particular sample time is used exclusively by explicit tasks. While creating these empty tasks is inefficient, it does not significantly impact the performance of the system. For users needing to eliminate these inefficiencies, the empty function and task may simply be deleted from the final code manually.

Finally, the most difficult challenge to the current code is the question of the what the priority level of implicit tasks should be. The problem lies in the limited amount of information available when generating the task code to run a particular implicit task. As the implicit tasks are defined by the rate at which they are run, it is possible to have both high priority and low priority code in the same task. The solution has been to recommend the user to limit the use of implicit tasks to only those blocks needed for inter-task communication since all implicit tasks are run as high priority. While this works well for the inter-task communication blocks, there isn't a way to enforce this rule within Simulink. However, it is not necessarily clear that it would be desirable to obey this rule in all cases.

Recognising these challenges it is shown here that automatic code generation has potential to accelerate research and development work with control systems. In particular, this work is suited to the kinds of control systems that would normally be run on an 8-bit AVR. While there are limits, they do not prevent the tool from being an effective rapid prototyping tool. And while the generated code may not be as efficient as code generated by a human expert, it still allows control system designers with minimal understanding of the underlying embedded system to access the benefits of a real time operating system.

Bibliography

[1] Jinhui Hu; Dabin Hu; Jianbo Xiao. Study of real-time simulation system based on rtw and its application in warship simulator. *"Electronic Measurement & Instruments"*, pages pp.3–966,3–970, Aug 2009.

[2] Embedded Coder Supported Hardware. `www.mathworks.com/products/embedded-coder/supported/index.html`, November 2012.

[3] Atmel Hardware Vendor. `http://www.windriver.com/products/bsp_web/bsp_vendor.html?vendor=Atmel&sort=6&order=1`, March 2014.

[4] ATmega640/1280/1281/2560/2561 Complete. /http://www.atmel.com/Images/doc2549.pdf, August 2012.

[5] Features Overview. `http://www.freertos.org/FreeRTOS_Features.html`, November 2012.

[6] DuinOS Project Home. `http://code.google.com/p/duinos/`, November 2012.

[7] Features Overview. `http://www.freertos.org/implementing-a-FreeRTOS-task.html`, November 2012.

[8] Chris Hote Tom Erkkinen. Automatic Flight Code Generation with Integrated Static Run-Time Error Checking and Code Analysis. *AIAA Modeling and Simulation Technologies Conference and Exhibit*, pages 1–2, 2006.

[9] Model Reference Simulation Targets. `http://www.mathworks.com/help/simulink/ug/model-reference-simulation-targets.html`, October 2013.

[10] File and Folder Created by Build Process. `http://www.mathworks.com/help/rtw/ug/files-and-folders-created-by-the-build-process.html`, November 2012.

[11] Develping S-Functions. `http://www.mathworks.com/help/pdf_doc/simulink/sfunctions.pdf`, September 2013.

[12] ISO/IEC 9899:1999. `http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?csnumber=29237`, September 2013.

[13] Michael Margolis. *Arduino Cookbook.* O'Reilly, 2012.

[14] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating system concepts.* John Wiley, 2009.

[15] ATmega640/1280/1281/2560/2561 Complete. `http://www.atmel.com/Images/doc2549.pdf`, October 2012.

[16] TDS2022B, TDS2024B Manual. `http://www.tek.com/oscilloscope/tds2022b-manual/tds2022b-and-tds2024b`, May 2008.

[17] CSTCE Series 16 MHz 2.5 X 2 mm 15 pF Built-in Capacitor SMT Resonator. http://search.murata.co.jp/Ceramy/image/img/w_hinm/L0430E.pdf, February 2014.

[18] Coding Conventions. http://www.freertos.org/Stacks-and-stack-overflow-checking.html, February 2014.

[19] Coding Conventions. http://www.mathworks.com/help/rtw/tlc/reading-record-files-with-tlc.html#bqlc5r9, November 2012.

Appendices

Appendix A

Code Compendium

A.1 arduino_srmain.tlc

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%% Abstract:
%%   Custom file processing to generate a "main" file.
%%
%% Copyright 2012 Joseph Moster
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%selectfile NULL_FILE

%function FcnFreeRTOSTaskingMain() void

  %if GenerateSampleERTMain
    %assign CompiledModel.GenerateSampleERTMain = TLC_FALSE
  %endif

  %assign cFile = LibCreateSourceFile("Source", "Custom", "ert_main")

  %openfile tmpBuf
  #include "%<LibGetMdlPubHdrBaseName()>.h"
  #include "WProgram.h"
  #include "DuinOS.h"
  %closefile tmpBuf
  %<LibSetSourceFileSection(cFile, "Includes", tmpBuf)>

  %openfile tmpBuf

  void __cxa_pure_virtual(void){}

  //Async = %<LibGetNumAsyncTasks()>
  //Sync = %<LibGetNumSyncPeriodicTasks()>
  //Task Count = %<LibGetNumTasks()>
  //Explicit Tasks = %<numTskBlks>
```

```
  //For each task
  %foreach tid = LibGetNumTasks()


    declareTaskLoop(stepTask%<tid>);
    taskLoop(stepTask%<tid>)
    {
      unsigned long previous = millis();
      %<LibCallModelStep(tid)>\
      delay(%<LibGetClockTickStepSize(tid)>*1000-(millis()-previous));
    }
  %endforeach

  int_T main(void)
  {
    /* Initialize model */
    init();

    %<LibCallModelInitialize()>\

    %foreach loopIdentifier = LibGetNumTasks()
                createTaskLoop(stepTask%<loopIdentifier>, HIGH_PRIORITY); //1
         %endforeach

  //Task Scheduler
  vTaskStartScheduler();

  //Will not get here unless a task calls vTaskEndScheduler():
  for (;;);

  //%<LibCallModelTerminate()>\
  return 0;
  }

  %closefile tmpBuf

  %<LibSetSourceFileSection(cFile, "Functions", tmpBuf)>

%endfunction
```

A.2 freeRTOS_task.tlc

```
%%
%% Abstract:
%%      TLC file for the freeROTS_task Block.
%%
%% Copyright 2012 Joseph Moster

%implements "freeRTOS_task" "C"
```

```
%% ================================================
%% Function: BlockTypeSetup
%%
%function BlockTypeSetup(block,system) void
  %% Create a global TLC variable to count number of Task blocks
  %assign ::numTskBlks = 0

%endfunction %% BlockTypeSetup

%% Function: BlockInstanceSetup ================================================
%% Abstract:
%%      Find and cache the needed Block records and names related to this block.
%%      Generate warnings if I/O not connected.
%%
%function BlockInstanceSetup(block, system) void
  %addtorecord block AsyncCallerGenCode TLC_TRUE

  %% Get the Block for the downstream f-c ss, warn and return if there is none
  %if LibIsEqual(SFcnSystemOutputCall.BlockToCall, "unconnected")
    %assign wrnTxt = "The output for FreeRTOS Task block '%<TaskBlockName>' is " ...
      "unconnected.  No code will be generated for this block."
    %<LibReportWarning(wrnTxt)>
    %return
  %endif

  %assign SSBlock = LibGetFcnCallBlock(block, 0)
  %addtorecord block ssBlock SSBlock

  %% Add Task's Name and information to the block
  %assign taskName = SFcnParamSettings.TaskName
  %assign taskPriority = SFcnParamSettings.TaskPriority
  %assign taskStackSize = SFcnParamSettings.TaskStackSize
  %assign block = block + taskName
  %assign block = block + taskPriority
  %assign block = block + taskStackSize

  %% Increment number of task blocks
  %assign ::numTskBlks = ::numTskBlks + 1
%endfunction

%% Function: Start =============================================================
%% Abstract:
%%      Create a task with the desired parameters
%%
%function Start(block, system) Output
  {
    %switch(taskPriority)
      %case 1
        createTaskLoopWithStackSize(%<taskName>_fcn, HIGH_PRIORITY,%<taskStackSize>);
        %break
      %case 2
```

```
        createTaskLoopWithStackSize(%<taskName>_fcn, NORMAL_PRIORITY,%<taskStackSize>);
        %break
      %case 3
        createTaskLoopWithStackSize(%<taskName>_fcn, LOW_PRIORITY,%<taskStackSize>);
        %break
      %default
        createTaskLoopWithStackSize(%<taskName>_fcn, NORMAL_PRIORITY,%<taskStackSize>);
        %break
    %endswitch
  }
%endfunction

%% Function: Outputs ============================================================
%% Abstract:
%%
%%
%function Outputs(block, system) Output
  %if !ISFIELD(block, "ssBlock")
    %return
  %endif

  %% I can't find any information on what the below code accomplishes
  %%if !block.GenCodeForTopAsyncSS
    %% Don't generate code for downstream f-c subsystem
    %% if GenCodeForTopAsyncSS is not set yet.
    %%return
  %%endif

  %% Call the downstream f-c subsystem, it can inline
  %openfile tmpBuf
  %<LibBlockExecuteFcnCall(block, 0)>\
  %closefile tmpBuf

  %openfile funcbuf

  //For each task
  declareTaskLoop(%<taskName>_fcn);
  taskLoop(%<taskName>_fcn)
  {
    unsigned long previous = millis();

    %if WHITE_SPACE(tmpBuf)
      /* Nothing to do for system: %<ssBlock.Name> */
    %else
      /* Call the system: %<ssBlock.Name> */
      %<tmpBuf>\
    %endif

        if(%<LibBlockSampleTime(block)>*1000>(millis()-previous))
        {
      delay(%<LibBlockSampleTime(block)>*1000-(millis()-previous));
        }
```

```
  }

  %closefile funcbuf
  %assign srcFile = LibGetModelDotCFile()
  %<LibSetSourceFileSection(srcFile, "Functions", funcbuf)>
%endfunction


%% [EOF] freeRTOS_task.tlc
```

A.3 freeRTOS_nicp.tlc

```
%% File : freeRTOS_nicp.tlc
%%
%% Description:
%%   Code generation file for freeRTOS_nicp

%% Copyright 2013 Joseph Moster

%implements freeRTOS_nicp "C"

%% Function: Outputs ====================================================
%%
%function Outputs(block, system) Output
  %if !SLibCodeGenForSim()
          %assign varIN = LibBlockInputSignal(0, "", "", 0)
      %assign varOUT = LibBlockOutputSignal(0, "", "", 0)

          taskENTER_CRITICAL();
      %<varOUT> = %<varIN>;
          taskEXIT_CRITICAL();
  %endif
%endfunction


%% [EOF]
```

Appendix B

Unit Testing

B.1 tests

```
%Copyright 2012 Joseph Moster
% Generates an html report based on the result of 19 Unit tests

filename = './testResults.html';
fid = fopen(filename,'w');

passMsg = 'APM not connected';
failMsg = 'Simulink:SFunctions:SFcnErrorStatus';

fprintf(fid,'<!DOCTYPE html>\n');
fprintf(fid,'<html>');

fprintf(fid,'<head>');
fprintf(fid,'<Thesis Verification Test>');
fprintf(fid,'</head>');

fprintf(fid,'<body>');

%% Priority Test Suite
taskName = 'highTask';
taskStack = 85;

fprintf(fid,'<H2>Priority Tests</H2>');

priority = 1;
[err,msg] = runTest();
pass = strcmp(err,passMsg);
writeResult(fid,strcat('Priority = ',num2str(priority)),pass,err,msg);

priority = 0;
[err,msg] = runTest();
pass = strcmp(err,failMsg);
writeResult(fid,strcat('Priority = ',num2str(priority)),pass,err,msg);
```

```
priority = 4;
[err,msg] = runTest();
pass = strcmp(err,failMsg);
writeResult(fid,strcat('Priority = ',num2str(priority)),pass,err,msg);

priority = -2;
[err,msg] = runTest();
pass = strcmp(err,failMsg);
writeResult(fid,strcat('Priority = ',num2str(priority)),pass,err,msg);

priority = 1.1;
[err,msg] = runTest();
pass = strcmp(err,failMsg);
writeResult(fid,strcat('Priority = ',num2str(priority)),pass,err,msg);

priority = 1+1i;
[err,msg] = runTest();
pass = strcmp(err,failMsg);
writeResult(fid,strcat('Priority = ',num2str(priority)),pass,err,msg);

priority = [1,2];
[err,msg] = runTest();
pass = strcmp(err,failMsg);
writeResult(fid,strcat('Priority = ',num2str(priority)),pass,err,msg);


%% Stack Size Test Suite
taskName = 'highTask';
priority = 1;

fprintf(fid,'<H2>Stack Size Tests</H2>');

taskStack = 80;
[err,msg] = runTest();
pass = strcmp(err,failMsg);
writeResult(fid,strcat('Task Stack Size = ',num2str(taskStack)),pass,err,msg);

taskStack = 5000;
[err,msg] = runTest();
pass = strcmp(err,failMsg);
writeResult(fid,strcat('Task Stack Size = ',num2str(taskStack)),pass,err,msg);

taskStack = -90;
[err,msg] = runTest();
pass = strcmp(err,failMsg);
writeResult(fid,strcat('Task Stack Size = ',num2str(taskStack)),pass,err,msg);

taskStack = 90+10i;
[err,msg] = runTest();
pass = strcmp(err,failMsg);
writeResult(fid,strcat('Task Stack Size = ',num2str(taskStack)),pass,err,msg);
```

```
taskStack = [90,100];
[err,msg] = runTest();
pass = strcmp(err,failMsg);
writeResult(fid,strcat('Task Stack Size = ',num2str(taskStack)),pass,err,msg);

taskStack = 89.3;
[err,msg] = runTest();
pass = strcmp(err,failMsg);
writeResult(fid,strcat('Task Stack Size = ',num2str(taskStack)),pass,err,msg);

%% Task Name Test Suite
taskStack = 85;
priority = 1;

fprintf(fid,'<H2>Task Name Tests</H2>');

taskName = 'a13characters';
[err,msg] = runTest();
pass = strcmp(err,failMsg);
writeResult(fid,strcat('Task Stack Size = ',num2str(taskName)),pass,err,msg);

taskName = 'a12character';
[err,msg] = runTest();
pass = strcmp(err,passMsg);
writeResult(fid,strcat('Task Stack Size = ',num2str(taskName)),pass,err,msg);

taskName = 'a btask';
[err,msg] = runTest();
pass = strcmp(err,failMsg);
writeResult(fid,strcat('Task Stack Size = ',num2str(taskName)),pass,err,msg);

taskName = 'a!=:;"@#$%^';
[err,msg] = runTest();
pass = strcmp(err,failMsg);
writeResult(fid,strcat('Task Stack Size = ',num2str(taskName)),pass,err,msg);

taskName = '1task';
[err,msg] = runTest();
pass = strcmp(err,failMsg);
writeResult(fid,strcat('Task Stack Size = ',num2str(taskName)),pass,err,msg);

taskName = '';
[err,msg] = runTest();
pass = strcmp(err,failMsg);
writeResult(fid,strcat('Task Stack Size = ',num2str(taskName)),pass,err,msg);

%% Close
fprintf(fid,'</body>');
fprintf(fid,'</html>');

fclose(fid);
```

```
%open(filename);
```

B.2 runTest.m

```
function [ err, output_string ] = runTest()
%UNTITLED2 Summary of this function goes here
%   Detailed explanation goes here


err = 'No Error';
output_string = 'Code Generation Successfull';
try
    slbuild nullTest;
catch e
    try
        if(e.identifier =='RTW:makertw:makeHookError')
            err = 'APM not connected';
            output_string = 'Code Generation Successfull';
        end
    catch e2
        err = e.identifier;
        output_string = e.message;
    end
end

end
```

B.3 writeResult.m

```
function [] = writeResult(fid,testName,pass,err,msg)
fprintf(fid,'%s<br>\n',testName);
if(pass)
    fprintf(fid,'<FONT COLOR="00ff00"><b>PASS</b></FONT><br>\n');
else
     fprintf(fid,'<FONT COLOR="ff0000"><b>FAIL</b></FONT><br>\n');
end
fprintf(fid,'\tError = %s<br>\n',err);
fprintf(fid,'\tMessage = %s<br>\n<br>\n',msg);
end
```

B.4 testResults.html

B.4.1 Priority Tests

Priority =1
PASS
Error = APM not connected
Message = Code Generation Successfull

Priority =0
PASS
Error = Simulink:SFunctions:SFcnErrorStatus
Message = Error reported by S-function 'freeRTOS_task' in 'nullTest/S-Function': Task priority parameter not in range 1-3

Priority =4
PASS
Error = Simulink:SFunctions:SFcnErrorStatus
Message = Error reported by S-function 'freeRTOS_task' in 'nullTest/S-Function': Task priority parameter not in range 1-3

Priority =-2
PASS
Error = Simulink:SFunctions:SFcnErrorStatus
Message = Error reported by S-function 'freeRTOS_task' in 'nullTest/S-Function': Task priority parameter not in range 1-3

Priority =1.1
PASS
Error = Simulink:SFunctions:SFcnErrorStatus
Message = Error reported by S-function 'freeRTOS_task' in 'nullTest/S-Function': Task priority parameter not an integer

Priority =1+1i
PASS
Error = Simulink:SFunctions:SFcnErrorStatus
Message = Error reported by S-function 'freeRTOS_task' in 'nullTest/S-Function': Task priority parameter invalid data type

Priority =1 2
PASS
Error = Simulink:SFunctions:SFcnErrorStatus
Message = Error reported by S-function 'freeRTOS_task' in 'nullTest/S-Function': Task priority parameter check failed

B.4.2 Stack Size Tests

Task Stack Size =80
PASS
Error = Simulink:SFunctions:SFcnErrorStatus
Message = Error reported by S-function 'freeRTOS_task' in 'nullTest/S-Function': Task stack size too small

Task Stack Size =5000
PASS
Error = Simulink:SFunctions:SFcnErrorStatus
Message = Error reported by S-function 'freeRTOS_task' in 'nullTest/S-Function': Task stack size too large (exceeds 4096 bytes)

Task Stack Size =-90
PASS
Error = Simulink:SFunctions:SFcnErrorStatus
Message = Error reported by S-function 'freeRTOS_task' in 'nullTest/S-Function': Task stack size too small

Task Stack Size =90+10i
PASS
Error = Simulink:SFunctions:SFcnErrorStatus
Message = Error reported by S-function 'freeRTOS_task' in 'nullTest/S-Function': Task stack size parameter invalid data type

Task Stack Size =90 100
PASS
Error = Simulink:SFunctions:SFcnErrorStatus

Message = Error reported by S-function 'freeRTOS_task' in 'nullTest/S-Function': Task priority parameter check failed

Task Stack Size =89.3
PASS
Error = Simulink:SFunctions:SFcnErrorStatus
Message = Error reported by S-function 'freeRTOS_task' in 'nullTest/S-Function': Task stack size parameter not an integer

B.4.3 Task Name Tests

Task Stack Size =a13characters
PASS
Error = Simulink:SFunctions:SFcnErrorStatus
Message = Error reported by S-function 'freeRTOS_task' in 'nullTest/S-Function': Error reading Task Name parameter, name is too long.

Task Stack Size =a12character
PASS
Error = APM not connected
Message = Code Generation Successfull

Task Stack Size =a btask
PASS
Error = Simulink:SFunctions:SFcnErrorStatus
Message = Error reported by S-function 'freeRTOS_task' in 'nullTest/S-Function': Task Name parameter contains non alphanumeric

Task Stack Size =`a!=:;\lq%@#$%^`
PASS
Error = Simulink:SFunctions:SFcnErrorStatus
Message = Error reported by S-function 'freeRTOS_task' in 'nullTest/S-Function': Task Name parameter contains non alphanumeric

Task Stack Size =1task
PASS

Error = Simulink:SFunctions:SFcnErrorStatus
Message = Error reported by S-function 'freeRTOS_task' in 'nullTest/S-Function': Task Name parameter does not have letter as first character

Task Stack Size =
PASS
Error = Simulink:SFunctions:SFcnErrorStatus
Message = Error reported by S-function 'freeRTOS_task' in 'nullTest/S-Function': Task Name parameter must be at least 1 character long

Appendix C

User Guide

C.1 General Setup

To use the freeRTOS software and libraries it is necessary to first install the tool. Once this has been completed there are a few simple steps required for each new Simulink model to use the tool.

C.1.1 Installation

How to add the real-time blockset to simulink:

- Install Arduino Simulink Library
- Install DuinOS
- Modify DuinOS file structure by moving the contents of the arduino/DuinOS folder into its parent folder
- Open Matlab and use the arduino.Prefs.setBoard command to view the available board types. Select the board being used that also has "_DuinOS" appended at the end of its name.
- Overwrite arduino_make_rtw_hook.m and arduino_srmain.tlc with the patch files included in the freeRTOS simulink library
- Add the FreeRTOS blockset by adding it's files to the Matlab path. This can be completed using the "Set Path" menu item.
- If the ArdupilotMega hardware is being used, Add the APM blockset to the Matlab path.

C.1.2 Configuration

How to setup a new simulink model with the real-time OS included:

- Open a new simulink model
- Under Solver/Solver Options set Type to "Fixed-step"
- Under Code Generation set the System target file to arduino.tlc.
- Add tasks as described below.

By double-clicking on a Free_RTOS task block, the user can view its mask as shown in Figure 3.2. The "Task name" field is the function name that will be used for the corresponding explicit function in the generated code. This name should be unique from all other task names in the model. The "Task Priority" drop-down allows the user to select between "Low", "Normal", and "High" priority levels. By selecting "User Defined" instead of "Minimum Stack Size" the user can then enter the desired stack size in the "Size" field. This capability is particularly important for large functions with many temporary variables. Having an insufficient stack size can cause the generated software to crash.

Double-clicking the "Function-Call Subsystem" block opens a sub-model diagram. The Function-Call Subsystem holds the block diagram that is executed by the Function-Call. In Figure C.1 a simple diagram from such a block is shown. In the model a "function" block is located above a diagram used to toggle a LED on pin 35.

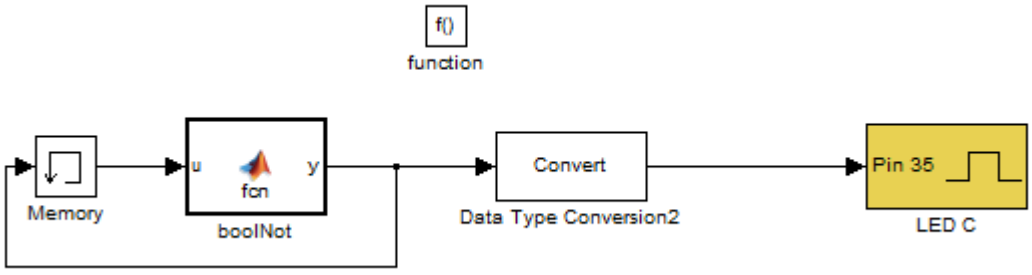

Figure C.1: An example of a simple explicit function

The options for the "function" block, shown in Figure C.2, should have the "Sample time type" set to periodic. The sample time of the function can then be set. This is the period that the generated code will try to execute the code.

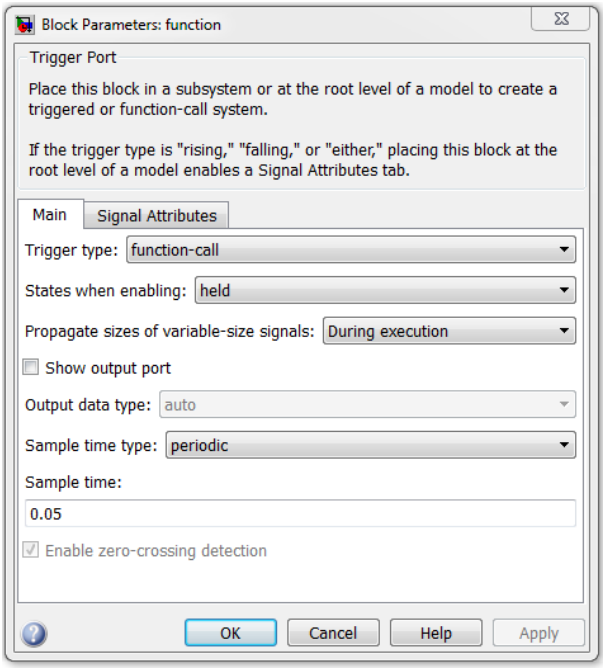

Figure C.2: Block parameters for the function block

Appendix D

Code Walk-through

This section provides a detailed step-by-step analysis of how the core code of this work functions. The code examined here is used to generate the actual C code and is written in the TLC format. There are a number of good resources on TLC syntax.[19] A minimalist explanation is included below as a reference.

- "%" Evaluate line as TLC commands
- "%<expr>" expr will be evaluated by the TLC
- "%%" a comment that will not be included in the generated file
- "//" a comment that will be included in the generated file

Lines without these special symbols are written straight to the file being generated.

D.1 Implicitly Defined Tasks and ert_main.c

The code needed to define implicit tasks and initilize the system are defined within the same C file. This file is generated by the arduino_srmain.tlc file whose complete contents can be found in Appendix A.1. This file defines a single TLC function called FcnFreeRTOSTaskingMain() that will be called during the build process by arduino_file_process.tlc. This function first creates a new C file called ert_main is created to house the generated code.

```
%assign cFile = LibCreateSourceFile("Source", "Custom", "ert_main")
```

Next, a new buffer is opened to hold the include statements. The model file header is included in addition to the standard Arduino and DuinOS headers.

```
%openfile tmpBuf
#include "%<LibGetMdlPubHdrBaseName()>.h"
```

```
#include "WProgram.h"
#include "DuinOS.h"
%closefile tmpBuf
%<LibSetSourceFileSection(cFile, "Includes", tmpBuf)>
```

A foreach statement sets up each implicit task. First a forward declaration of each task is made using the "declareTaskLoop" macro. Implicit tasks are given the name "stepTask" and an incremented integer. The "taskLoop" macro then creates an appropriately named function with an infinite loop containing the specified code. The contained code records the time, calls the task's function, and then waits till the next time the code needs to run. The TLC function %<LibCallModelStep(tid)> writes the code needed to call the function.

```
//For each task
%foreach tid = LibGetNumTasks()
  declareTaskLoop(stepTask%<tid>);
  taskLoop(stepTask%<tid>)
  {
    unsigned long previous = millis();
    %<LibCallModelStep(tid)>\
    delay(%<LibGetClockTickStepSize(tid)>*1000-(millis()-previous));
  }
%endforeach
```

An example of the code generated by this function is shown below.

```
declareTaskLoop(stepTask0);
taskLoop(stepTask0)
{
  unsigned long previous = millis();
  IMU_Tasked_step0();

  /* Get model outputs here */
  delay(0.025*1000-(millis()-previous));
}
```

Once the tasks have been declared, the "main" function can now be declared. This main function first calls the Arduino's init() function. Then the TLC function %<LibCallModelInitialize()> generates a line of code to call the model's initialization function. The model's initialization function will handle the declaration and setup of explicit tasks. The main function then adds each implicit task to the scheduler and then starts the task scheduler. An infinite loop is then declared, however the task scheduler will not return unless stopped. The ability to stop the controller is not made available to user, so execution should never reach this infinite loop.

```
int_T main(void)
{
  /* Initialize model */
  init();
```

```
    %<LibCallModelInitialize()>\

    %foreach loopIdentifier = LibGetNumTasks()
        createTaskLoop(stepTask%<loopIdentifier>, HIGH_PRIORITY); //1
    %endforeach

    //Task Scheduler
    vTaskStartScheduler();

    //Will not get here unless a task calls vTaskEndScheduler():
    for (;;);

    //%<LibCallModelTerminate()>\
    return 0;
  }
```

An example of the generated main function shown below illustrates how the code generated is actually more simple than the code that generated it.

```
int_T main(void)
{
  /* Initialize model */
  init();
  IMU_Tasked_initialize();
  createTaskLoop(stepTask0, HIGH_PRIORITY);//default
  createTaskLoop(stepTask1, HIGH_PRIORITY);//default

  //Task Scheduler
  vTaskStartScheduler();

  //Will not get here unless a task calls vTaskEndScheduler():
  for (;;) ;

  //    IMU_Tasked_terminate();
  return 0;
}
```

Now that code generation for this file is complete, the buffer is closed and added to the ert_main file.

```
  %closefile tmpBuf
  %<LibSetSourceFileSection(cFile, "Functions", tmpBuf)>

%endfunction
%%[EOF]
```

D.2 Explicitly Defined Tasks

Explicitly defined tasks are defined as a part of the larger C file that is generated for the model. As such, instead of re-defining the entire structure of the document, it is only necessary to define

how to handle these blocks when code generation asks them to return their corresponding lines of code. This is accomplished with the freeRTOS_task.tlc file which is examined in detail here.

The "Start" function of the freeRTOS_task block defines what code will be added to the model's initialization function. The variables "taskPriority", "taskStackSize", and "taskName" are special values available from the freeRTOS_task's RTW file. These are the values that were added to the RTW file by the Mex files from the user interface. Based on the priority set by the user, the correct line of code is added by the switch statement.

```
%function Start(block, system) Output
  {
    %switch(taskPriority)
      %case 1
        createTaskLoopWithStackSize(%<taskName>_fcn, HIGH_PRIORITY,%<taskStackSize>);
                %break
      %case 2
        createTaskLoopWithStackSize(%<taskName>_fcn, NORMAL_PRIORITY,%<taskStackSize>);
        %break
      %case 3
        createTaskLoopWithStackSize(%<taskName>_fcn, LOW_PRIORITY,%<taskStackSize>);
        %break
      %default
        createTaskLoopWithStackSize(%<taskName>_fcn, NORMAL_PRIORITY,%<taskStackSize>);
        %break
    %endswitch
  }
%endfunction
```

The "Outputs" function writes code for declaring a new task into the "funcbuf" function buffer. This buffer is then written to the Functions portion of the "srcFile". This puts the created function into the C model file.

To create the task code, the code that the task will execute is first stored in the "tmpBuf" buffer. This buffer's contents will later be added by the line "%<tmpBuf >" The function buffer is then opened and a new taskLoop declaration is added. The task is then constructed much like the implicit function. The time is recorded, the task's function is run, and then the function is put to sleep until it needs to run again.

```
%function Outputs(block, system) Output

  ... Some error checking code omitted for clarity

  %% Call the downstream f-c subsystem, it can inline
  %openfile tmpBuf
  %<LibBlockExecuteFcnCall(block, 0)>\
  %closefile tmpBuf

  %openfile funcbuf
```

```
  //For each task
  declareTaskLoop(%<taskName>_fcn);
  taskLoop(%<taskName>_fcn)
  {
    unsigned long previous = millis();

    %if WHITE_SPACE(tmpBuf)
      /* Nothing to do for system: %<ssBlock.Name> */
    %else
      /* Call the system: %<ssBlock.Name> */
      %<tmpBuf>\
    %endif

        if(%<LibBlockSampleTime(block)>*1000>(millis()-previous))
        {
      delay(%<LibBlockSampleTime(block)>*1000-(millis()-previous));
        }
  }

  %closefile funcbuf
  %assign srcFile = LibGetModelDotCFile()
  %<LibSetSourceFileSection(srcFile, "Functions", funcbuf)>
%endfunction
```

The remainder of the TLC code simply closes the function buffer and adds the code to the library of functions that will be incorporated into the final C file for the model.