Revision: September 23, 2014

Author: Professor Richard Wall, University of Idaho, rwall@uidaho.edu

1300 NE Henley Court, Suite 3
Pullman, WA 99163
(509) 334 6306 Voice | (509) 334 6300 Fax

Project 11: FreeRTOS™ Real-Time Control of a Stepper Motor

Table of Contents

Purpose

This project consists of a series of reference designs that will teach the fundamentals if real-time operating systems for embedded control. The each of 12 reference designs builds on previous projects with increase in complexity. The initial reference introduce methodologies and standard terminology used in RTOS based embedded system design.

Minimum Knowledge and Programming Skills

1. Knowledge of C or C++ programming
2. Working knowledge of MPLAB ® X IDE
3. Understanding embedded system design

Equipment List

1. chipKIT™ Pro MX7 processor board with USB cable
2. PmodSTEP Stepper Motor Driver module or
3. Microchip MPLAB ® X IDE
4. MPLAB ® XC32 Compiler

Software Resources

1. XC32 C/C++ Compiler Users Guide
2. MPLAB ® X User's Guide
3. C Programming Reference

References

1. chipKIT™ Pro MX7 Board Reference Manual
2. chipKIT™ Pro MX7 Board Reference Schematic
3. FreeRTOS Quick Start Guide
4. FreeRTOS Customization Parameters
5. Introduction to RTOS

Real Time Operating Systems Introduction

In general, an operating system (OS) is responsible for managing the hardware resources of a computer and hosting applications that run on the computer. Modern microprocessors have sufficient memory capacity and performance to host one or more complex applications comprised of multiple tasks that require the sharing of critical resources such as human-machine interfaces (keypads and buttons, and displays) communications, memory, and

processor time. With the proper computer software, the processor is able to switch tasks so frequently and rapidly that it appears that a single processor is performing the tasks in parallel. A real-time operating system (RTOS) is a library of functions designed as a management program that allocates the processors resources such that the system performance meets specific timing requirements without conflicts between independent tasks. The advantages offered by a professional grade RTOS include assurance that time critical elements of the program meet the timing requirements, promotes system partitioning into tasks that are easier to develop and maintain, and allows the assignment

The claim that every embedded application has an OS at the core of its operation is justified in light of the functional requirements of an OS. When the application has a firm requirement for time performance, then the OS must now incorporate the elements of RTOS. Many embedded applications have minimal requirements for management of resources because of low demand for processor resources. However, the complexity of a software-based system can quickly expand and the resource management requirements exceed the performance provided by polling tasks in a round-robin fashion or a demand base system that uses preemption (interrupts). One of the challenges to embedded system design is determining when to transition for an in-house developed OS and making the needed investment to use a commercially available OS.

When making this decision one must always be conscious of the fact that commercial RTOS packages rarely reduce the memory requirements or increase the operating speed. One can rightly ask, then why bother with a commercial system? The answer lies in two areas: development time and time needed for code validation. Commercial RTOS systems have undergone many hours of software testing both by the developers of the RTOS and by those who integrate the RTOS in to their own applications. Most times the RTOS library of functions is considered SOUP – Software of Unknown Providence. Commercial RTOS software packages provide a systematic and dependable approach to resource management designed to guarantee critical timing is maintained.

The popularity of Free RTOS as an off the shelf (OTS) RTOS[1] is due to the fact that it free. Although it does not have all of the features of a full featured RTOS, it provides basic support for multiple threads or tasks, queues, mutexes, semaphores and software timers. FreeRTOS™ currently supports 34 different microprocessors and 18 different tool chains or development environments. Although the basic RTOS is in fact free, there is a cost for documentation targeting a specific tool chain and processor. There is also a cost for additional features.

Various algorithms are employed to schedule processor tasks. The two most common are the cooperative and preemptive. Cooperative multitasking depends on a task relinquishing their control of the processor and generally works best when one or more of the tasks sends considerable time waiting for an external event (such as the arrival of a communications packet) or internal timed event. For cooperative multitasking scheduling to meet the requirements for real-time, the accumulated time to execute all tasks must be less than the shortest period required to repeat a given task. Preemptive scheduling is based upon processor interrupts such as a clock timer or the occurrence of an external event. Preemptive scheduling offers the advantage of allowing priorities to be assigned based upon the need for responsiveness. It is common for an RTOS to use both cooperative and preemptive scheduling in a single application.

A thread or task can be thought of as a job or an operation. Some tasks consist of a collection of smaller tasks. Each task is scheduled and is either ready to run, running or blocked. Communications between tasks use queues, semaphores, and mutexs (mutually

[1] http://en.wikipedia.org/wiki/Real-time_operating_system

exclusive semaphore). Queues are generally used to pass information while semaphores are used to synchronize tasks and are binary in nature (i.e. either TRUE of FALSE).

Essential Elements of FreeRTOS

Definitions

Execution – the performance of instructions in a computer program

Concurrency – the property of a system that supports multiple simultaneous computations

Context – the state of the processor as defined by the values stored in the program counter, stack pointer, CPU registers, status flags and selected global variables

Kernel – the part that schedules which thread gets to execute at a given point in time

Multithreading – the ability to manage a processor so that it executes code in multiple threads

Multitasking – the ability to switch between different tasks that give the appearance of managing all tasks simultaneously.

Operating System – software that manages hardware and software resources to provide services for computer programs

Process – an overall context in which the thread runs

Program – an executable file

Task – a specific piece of work to be done – usage varies: in computing, a **task** is a unit of execution. In some operating systems, a task is synonymous with a process, in others with a thread. In batch processing computer systems, a task is a unit of execution within a job.

Thread – a single sequential flow of control

Concepts

FreeRTOS is a collection software library files that contain functions called by a user application. The files are organized in a manner that allows MPLAB to build a project. The file structure is described in Appendix A. RTOS consists of three major components: application hardware initialization, one or more tasks, and a scheduler that determines which task is to be running. Some applications can also require independent preemption tasks that are not under the control of the RTOS scheduler but in many cases must communicated with tasks that are under the control of the scheduler.

Programs written for embedded system applications follow a typical format: hardware and software resources are initialized followed by an infinite loop containing the code to implement that application processes. Applications developed using a RTOS break with this traditional format. The order of operations starts with the general hardware initialization. This includes

initializing interrupt that not managed by the RTOS scheduler. Next each task is created using a function call to *xTaskCreate* that registers each task with the scheduler along with parameters that the scheduler passes to the task. Queues (*xQueueCreate*) and semaphores (*xSemaphoreCreateMutex*) are created that provide communications between tasks using global variables. Finally, the function, *vTaskScheduler,* is called from which the code execution never returns.

The format of each task follows the general format of the typical *main* function. Local variables are declared and initialized if appropriate to do so. This is followed by code that is to be executed only once such as initialization of specific hardware and variables. Finally, each task contains an infinite loop from containing the task functionality which the task is never to exit. The scheduler perceives each task in one of three states illustrated in Figure 1. Each task is initially in the Ready state and is waiting for the scheduler to start running the task. Since each task is provided with its own stack memory space, static local variables should not be declared with in a task function.

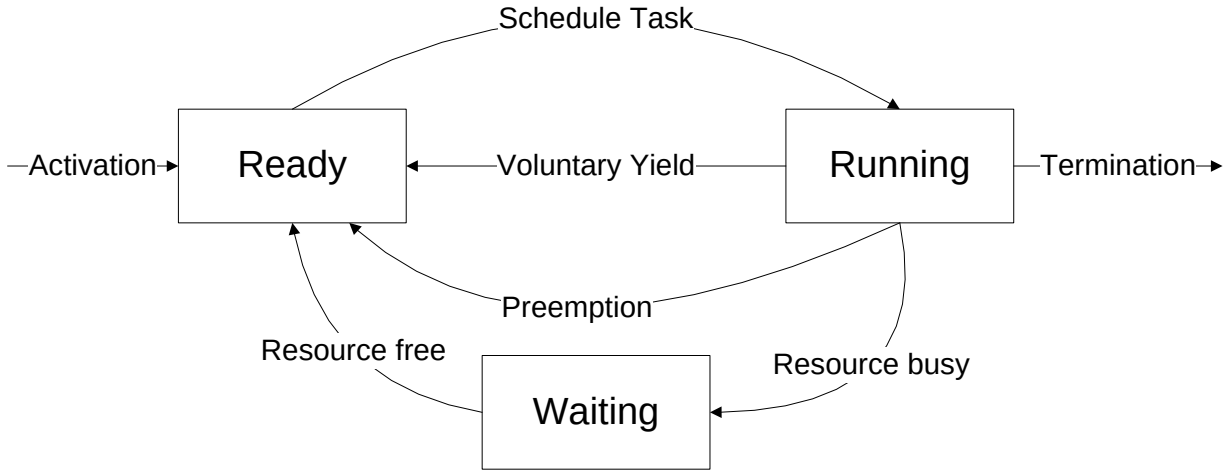

Figure 1. Task states and transitions

There are four different ways to stop a task. A task can terminate itself in which by removing the task from the scheduler never to be executed again. A task can be "blocked" by waiting for a resource to become free or available. In this case, the task transitions to the Waiting state. A task can be preempted by the scheduler when the scheduler determines that a task with a higher priority is ready to run or it is time to share the execution time with a task that is of the same priority level. Finally, a task can voluntarily return to the scheduler on its own accord.

A task that has transitioned to the waiting state remains in that state until the resource that was blocking the execution becomes available. Examples of a blocking resource is waiting for a button to be pressed and waiting for serial communications. Tasks can also be waiting for a time delay to expire. As such, the RTOS can only guarantee a minimum time delay. But because another task may be running either at the same or higher priority level, the delay time may be extended.

As discussed above, schedulers use various algorithms to determine which task should be running. A schedule operating on fairness allocates equal time to all tasks that are ready to run and programmed for the same priority level. The execution time is divided into time slices that are defined by the RTOS preemptive tick clock. The determination is made as to which task to run next either at the end of each time slice or whenever a task exits the running state for one of the four reasons listed above. Tasks that have the highest priority are checked to see if any of those tasks are ready to run. If so, the scheduler runs those tasks. If not, then the scheduler

checks for tasks at the next lower priority level that are ready to run. This process continues until the scheduler identifies a task that is ready to run. If no tasks are ready to run, the scheduler runs an idle task. For additional information regarding the operation of the task scheduler, see the FreeRTOS web link for Real Time Scheduling at http://www.freertos.org/implementation/a00008.html.

FreeRTOS Reference Design Descriptions

The following reference designs are demonstrations of FreeRTOS™ operating systems being used to control simple functions. Each successive reference design increases in complexity and leverages for previous designs. The development environment uses MPLAB ® X version 1.85 or above and the XC C/C++ compiler version 1.2 or higher. The designs use FreeRTOS v7.5.2 and the chipKIT™ Pro MX7 board that is equipped with a PIC32MX795F512 processor.

The FreeRTOS library functions use the following naming conventions for their coding standard and style guide.

A. Variables

   i. Variables of type *char* are prefixed *c*
   ii. Variables of type *short* are prefixed *s*
   iii. Variables of type *long* are prefixed *l*
   iv. Enumerated variables are prefixed *e*
   v. Other types (e.g. structs) are prefixed *x*
   vi. Pointers have an additional prefixed *p*, for example a pointer to a short will have prefix *ps*
   vii. Unsigned variables have an additional prefixed *u*, for example an unsigned short will have prefix *us*, and a pointer to an unsigned short will have prefix *pus*.

B. Functions
   i. File private functions are prefixed with *prv* File private functions are prefixed with *prv*
   ii. API functions are prefixed with their return type, as per the convention defined for variables
   iii. Function names start with the file in which they are defined. For example *vTaskDelete* is defined in the tasks.c file, and has a void return type. A function prefix of *pv* is a pointer to a function that returns a *void*.

C. Macros
   i. Macros are pre-fixed with the file in which they are defined. The pre-fix is lower case. For example, configUSE_PREEMPTION is defined in FreeRTOSConifg.h.
   ii. Other than the pre-fix, macros are written in all upper case, and use an underscore to separate words.

RD1 – Simple Task Scheduler Example:

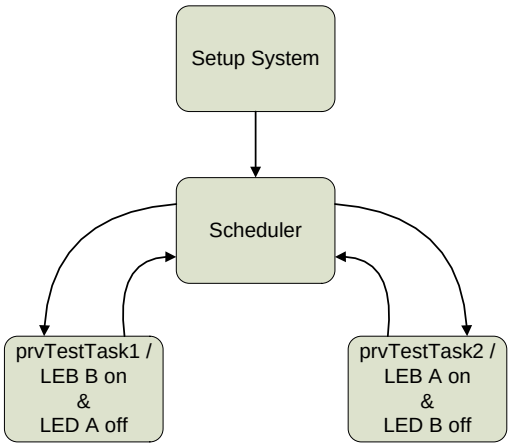

Figure 2. Control Flow Diagram for RD1

This project introduces the basic process of building an application under the FreeRTOS environment. The operating system creates manages two tasks that turn on an LED and increments a counter. The program continues to run the code in that function until the scheduler stops the task that is currently running and starts the other task,

Task1 turns LED A on and turns LED B off. Task2 turns LED B on and turns LED A off. Independent counters in each tasks is incremented each time a task turns on the associated LED. If a logic analyzer or oscilloscope is connected to the PmodSTEP test points for LED A and LED B, one will see two square waves that are 180 degrees out of phase from each other as shown in Figure 3. This indicates that each task is allotted one timer tick to execute. Since both tasks are assigned the same priority level and use all of their allotted task time, the scheduler implements a fair and equal distribution of operating time. No idle task is scheduled.

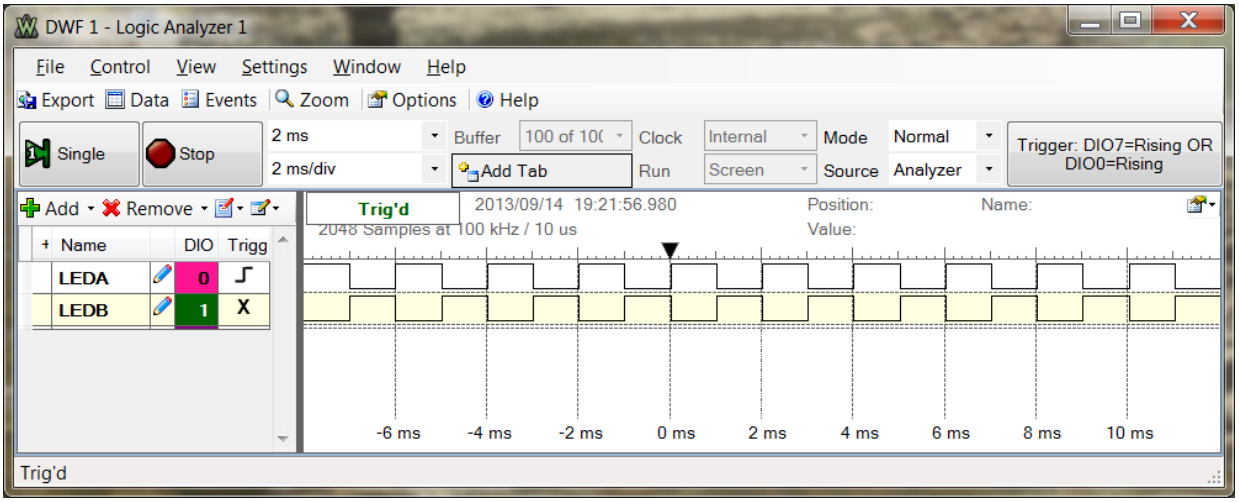

Figure 3. Screen capture showing the time each task is running for RD1.

RD2 - Multiple Tasks Implemented Using Common Execution Code

This reference design demonstrates how to use reentrant code to implement two tasks that use the code written as a single function. This is a common technique for minimizing the amount of required lines of code to manage tasks that essentially perform the same operation on different application resources. Examples of this include reading different push buttons or controlling different LEDs.

In this reference design, the operating system manages three tasks that turn on an LED and increments a counter. One task is scheduled twice and uses parameters passed from the scheduler to determine which LED to turn on. The operation uses the idle hook task to determine if the scheduler has any idle time. Since all tasks are set to the same priority as the idle task – they all get the same allotment of CPU time as shown in Figure 4. The idle task yields as soon as the execution of the function *vApplicationIdleHook* is completed. The operation uses the idle hook task to determine when the scheduler has idle time. The *configUSE_IDLE_HOOK* bit in *FreeRTOSconfig.h* must be set to a 1 and the function *vApplicationIdleHook* defined in the application code as illustrated in Listing 1 below. The expanded timing diagram shown in Figure 5 demonstrates that the idle hook task only requires 5us. The LED used for timing the idle time is reset in each task.

Listing 1.

```
void vApplicationIdleHook( void )
{
static unsigned long ulIdleCycleCount = 0UL;
    ulIdleCycleCount++;     /* System declared global variable */
    LATBSET = LEDH;         /* For timing instrumentation only */
}
```

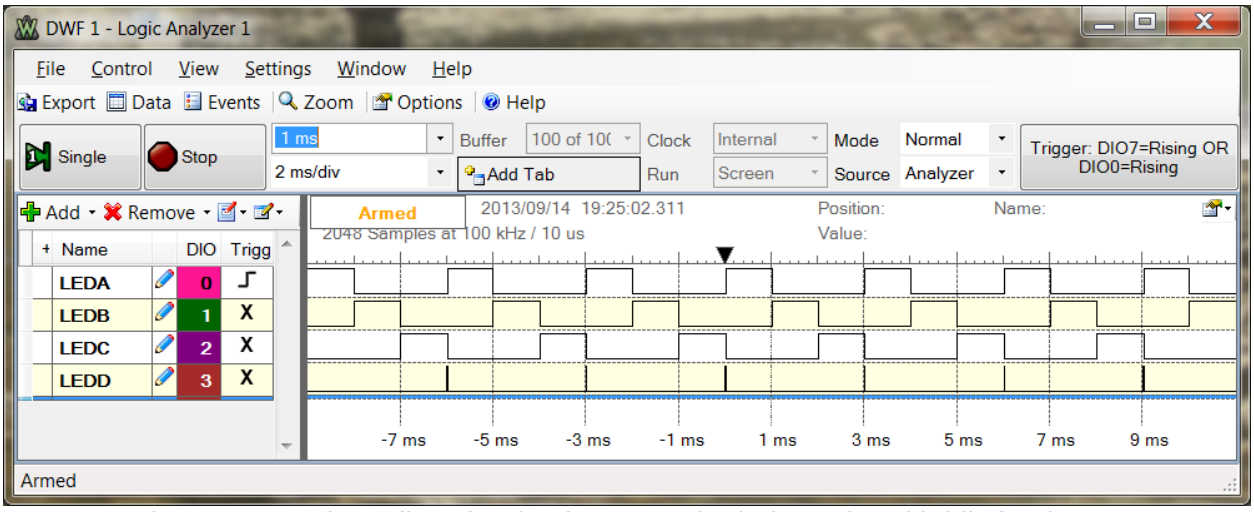

Figure 4. Run time allocation for three equal priority tasks with idle hook trap

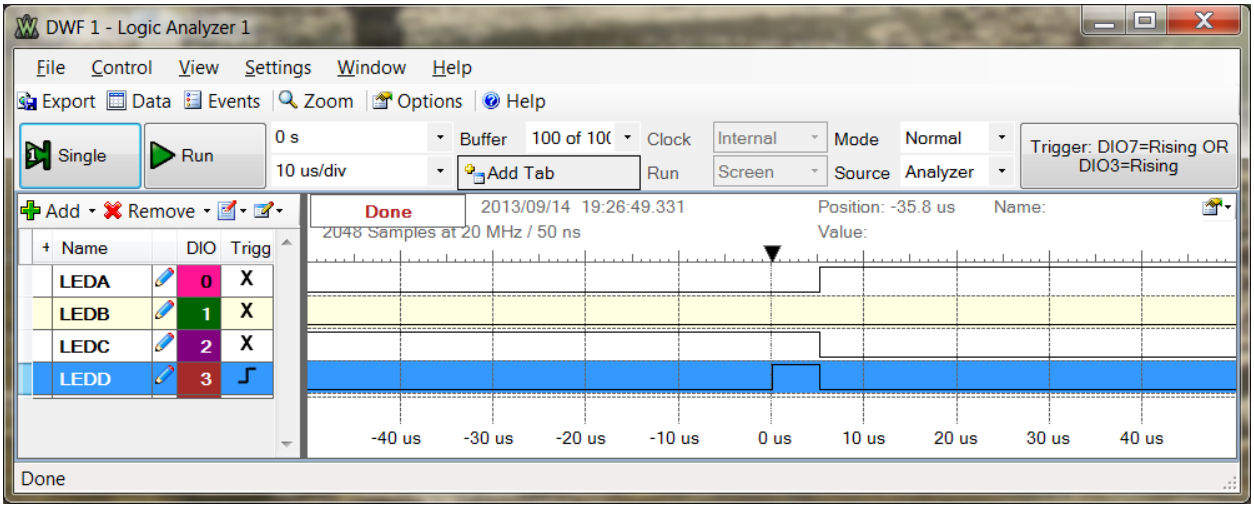

Figure 5. Execution time of the idle hook function

RD3 - Setting Task Priorities

This application reference design creates and manages multiple tasks that turn on an LED and increments a counter. Four tasks toggle LEDs at various rates. Both the *vTaskDelay* and *vTaskDelayUntil* function call block the tasks for various periods. The *prvTestJacob* task has no parameters passed to it by the scheduler.

The *prvTestCody* task toggles LEDD each millisecond. This task also uses the *vTaskDelayUntil* delay function that implements a period delay that is not dependent on the time to execute the task code. The *vTaskDelayUntil* function implements a true period delay rather than a fixed time delay implemented by *vTaskDelay.* The *prvTestCody* task is set for a higher priority level than the other tasks and toggles LEDD with a 100ms period.

The *prvTestBen* task is scheduled twice to control LEDB and LEDC. Each creation of the *prvTestBen task* uses a different set of task parameters that controls LEDB to toggle with a period of 250 ms and LEDC to toggle with a 750ms period. The two scheduled *BenTasks* demonstrate the potential for code conservation by using reentrant functions. The main reason it is possible to make the the code for the *prvTestBen* task reentrant is that each schedule of the task uses independent resources. In this case - the resources that are not shared are the LED that is being flashed and the time delay.

This application uses the idle hook task to determine when the scheduler has idle time. The *configUSE_IDLE_HOOK* bit in *FreeRTOSconfig.h* must be set to 1. Each time the idle hook function is called, LEDH is turned on. Whenever any other task resumes running, LEDH is turned off. Since LEDH is high (meaning the idle task is running) most of the time, very little time is spent executing the idle task code and execution is immediately passed on to the four scheduled tasks as shown in Figure 6.

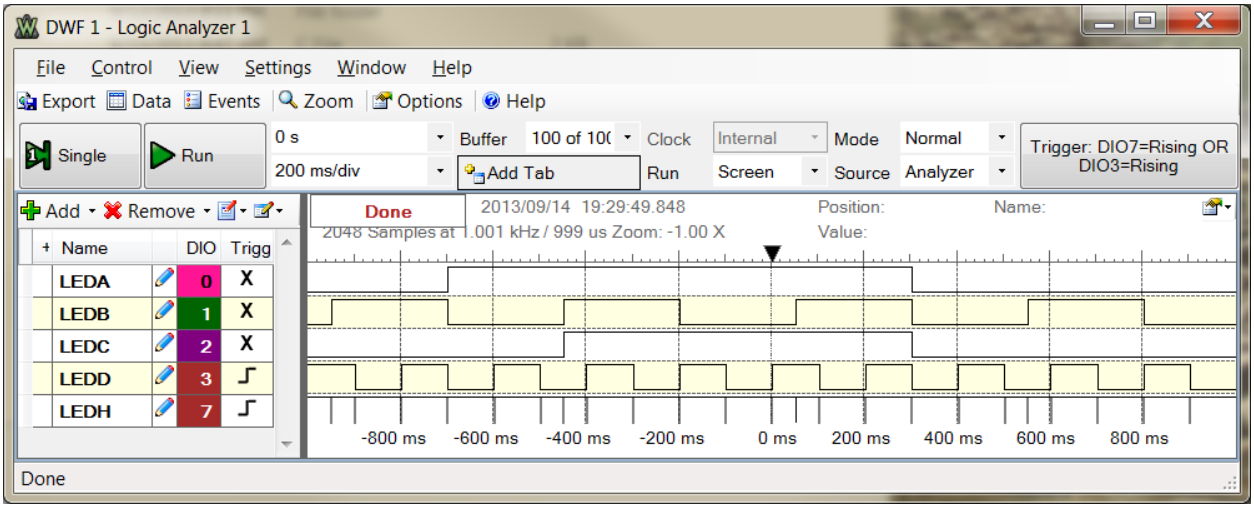

Figure 6. Four tasks running at various rates controlled by OS delay and delay until functions.

RD4 -Task Queues

The operating system manages three tasks that toggles an assigned LED and based upon which button is pressed. That task in turn sends a queue to another LED control function that changes the state of a second LED. The single button (prvbutton) function serves the three scheduled tasks to process the three different button inputs. The operation of each button task is determined by the argument values in pvParameters. The program uses a queue (xButtonQueue) to send a message from the button task to the LED (prvled) task to indicate which other LED to toggle. The xButtonQueue depth is set to 10 meaning that if a button is pressed faster than the LEDs cab be toggled, up to 10 press operations can stacked up and will eventually be processed.

The xButtonQueue task has parameters passed to it that indicates which button to monitor (port and bit), which PORT B LED to toggle and which PORT B LED to have the *prvled* task toggle.

The xButtonQueue task is created three times with parameters to specify BTN1, BTN2, and BTN3. The prvled task is pased parameter to indicate which PORT to use for toggling the LEDs. When BTN1 is pressed, the *xButtonQueue* task toggles LEDA each 100ms and sends a *qMwessage* to the *prvled* task to toggle LEDB. The *prvled* task has a delay of 250ms. Hence while BTN1 is pressed LEDA blinks on or off each 100ms indicating that a message is being sent to the prvled task. The prvled task is taking messages out of the queue slower than messages are being put in to the queue buffer. When the queue buffer is filled, both LED A and LEDB will blink at the 250 ms rate. This shows that as the prvled task reads a message from the queue buffer, memory is available for the prvled task to place another message into the queue buffer. Once BTN1 is released, LEDA stops blinking but LEDB continues to blink until the queue buffer is emptied. All three button tasks use the same message queue to pass data the prvled task. Hence holding down two or three buttons fills the queue buffer faster.

The above description applied to BTN2 that controls LEDC and LEDD and also BTN3 that controls LEDE and LEDF. See the declarations for Button1 through Button3 in main. This operation uses the idle hook task to determine when the scheduler has idle time. The

configUSE_IDLE_HOOK bit in *FreeRTOSconfig.h* must be set to 1. LEDH is set each time the OS enters the idle state and is turned off whenever the OS returns to an application task.

Note: This program DOES NOT perform a "press on - press off" operation. This functionality is left to the motivated student as an assignment.

RD5 – Stepper Motor Control Using Timer Interrupts

This program controls a stepper motor based upon the status of BTN1 and BTN2. The Button task (*prvButtons*) detects a change of button status, decodes the button controls, and determines the values of step delay, direction, and step mode. These values are passed via a queue to the stepper motor control task (*prvStepperStep*). The stepper motor control task is blocked from execution until a semaphore is sent from the Timer 3 ISR. The queue for messages from the button detection task is checked each step to determine if new operating parameters have been set by pressing BTN1 or BTN2.

The operation uses the idle hook task to determine when the scheduler has idle time. The *configUSE_IDLE_HOOK* bit in *FreeRTOSconfig.h* must be set to 1. LEDD is set each time idle hook is run. The button and stepper motor control task reset LEDD when resumed.

LEDA is toggled each time the Timer3 ISR sends a semaphore to the stepper motor step task (*prvStepperStep*). LEDB is toggled each time a step is taken. The button task checks for a button press then executes a 100ms delay using "*vTaskDelay*". The scheduler stays in the idle task and does not go back to the button task (prvButtons) until the delay period has expired or a semaphore is received by the stepper motor control task (*prvStepperStep*). Hence, most of the processors execution time is spent in the idle mode as illustrated in Figure 7**.**

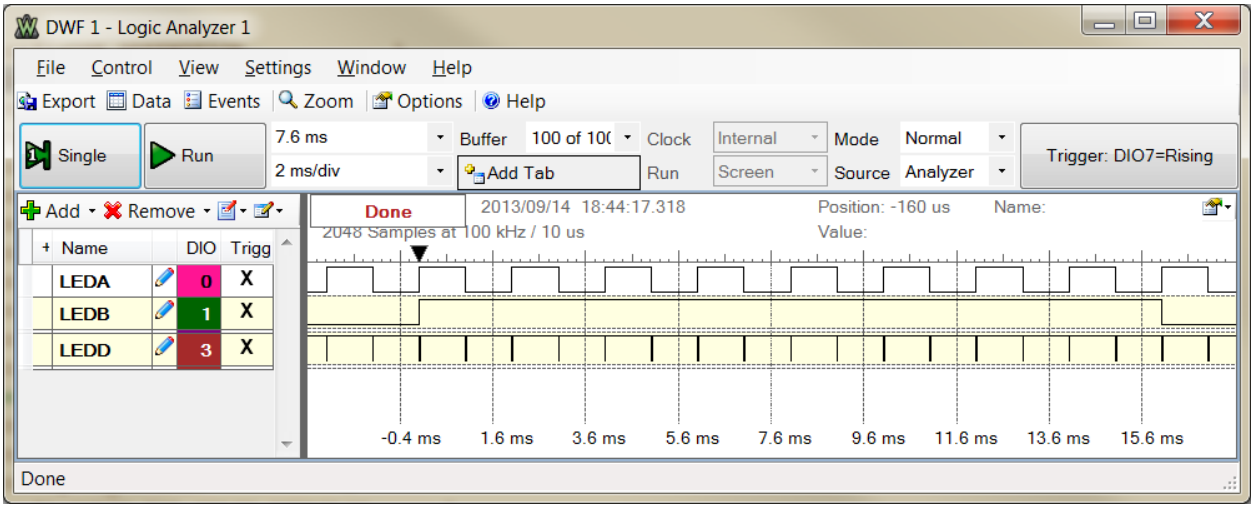

Figure 7. Stepper motor control timing with idle time.

A possible design modification is to control LED1 and LED2 when BTN1 and BTN2 are pressed.

RD6 – LCD Control Using Mutex Semaphores

In this reference design a counter values is passed between two tasks each task increments the counter before passing it back. Each task sends a message to the LCD that is protected by a mutually exclusive (mutex) semaphore. Commenting out the semaphore take and give instructions demonstrates how the LCD text gets messed up without using the device exclusion protection provided by the mutex.

RD6a - I2C EEPROM Control Using Mutex Semaphores

The purpose of this code is to program an I2Cwith 1024 bytes of randomly generated data starting at a random address and verify that the data was correctly saved. The specific lines of code that directly addresses each item in the specification list are identified in main.c. Access to the LCD and EEPROM are protected by “mutex” semaphores. LEDH is used for idle task timing only. LEDA is used to time the data generation task. LEDB is used to time EEPROM write operation. LEDC is used to time the read memory task.

Functional Specifications for RD6a I2C_EEPROM:

a. RD6a RTOS Tasks
   i. BUTTON_DET:
      1. Button press detect (BTN1)
      2. Operates in PUSH-ON PUSH-OFF
   ii. Toggles LED1
   iii. Send queue message to DATA_GEN to initiate test when button is pressed
b. DATA_GEN: Generates random data array and write EEPROM
   i. Waits for BUTTON_DET message
   ii. Turns on LEDA
   iii. Generates 1024 byte random array and random bytes (module 256) and random starting address (modulo 32768).
   iv. Turns on LEDB
   v. Writes starting address in hexadecimal to line one of the LCD (employes mutex semaphore #1.
   vi. Turns off LEDB
   vii. Sends 1024 bytes of random values and starting address to
c. DATA_CHK task using queue. (suggestion: consider structure to pass data)
   i. Waits for results from DATA_CHK task
   ii. Turns off LEDA
d. DATA_CHK: Reads EEPROM and compares with data sent from DATA_GEN.
   i. Waits for message queue from DATA_GEN task
   ii. Writes starting address in hexadecimal to line two of the LCD (employes mutex semaphore #1)
   iii. Turns on LEDC
   iv. Read EEPROM starting at address communicated by message

v. Turns off LEDC
vi. Compares 1024 bytes of data received from DATA_GEN with data read from EEPROM
vii. Writes either "PASSED" or "FAILED" to second line of LCD following the hexadecimal value of the starting address (employs mutex semaphore #1))

RD7 – UART Control Using Interrupts in an FreeRTOS Environment

This reference design demonstrates how to implement UART line base IO at 19200 BAUD. One task reads a character at a time and fills a buffer until a NL or CR character is detected or the buffer has reached its size limit. The message is then sent to a task that sends the text string back to the UART. RX and TX interrupts are used to manage character based serial communications. The serial communications uses UART 1 at 19200 BAUD N81.

Note: You will not see anything on the terminal screen while entering text until you press the enter key unless you have the terminal setup for local echo characters. Lines of text echoed back to the terminal have CR, and LF appended.

RD8 – TIMERS

This reference design uses the same serial code as RD7 for the serial communications. A timer is started that starts a task at a specific interval. The tick count is reported to the serial terminal each second. Timers.c and Timers.h must be added to this project

RD8a – Stepper Motor Control using RTOS Timer API

An example of FreeRTOS running on an chipKIT™ Pro MX7 using a PIC32MX7 processor. This example controls the stepper motor similar to RD 5 except that the RTOS timer API is used instead of a timer interrupt to control the speed of the stepper motor. The stepper motor step interval is changes using the *xTimerChangePeriod* statement in the prvButtons task. Note the additions to FreeRTOSConfigure.h for configurations necessary to use timers.

RD9 – Real Time Performance Statistics

This program implements the Stepper motor control problem using a RTOS. The stepper motor speed, direction and mode are controlled at 6 pre-defined operating points based upon the conditions of the BTN1, BTN2, and BTN3 controls. All buttons operate as push on - push off switches. The stepper motor can be controlled to speeds that resolve to whole milliseconds per step intervals from 0.1 RPM to 60 RPM. The serial port uses the PIC32 UART1 at 19200 BAUD. Statics are sent to the serial terminal using a serial Tx queue one line at a time whenever LED1, LED2 and LED3 are all off.

Notes: This version implements the vTaskGetRundTimeStats api that gathers the run time statistics and reports them whenever the buttons are sent to the 0 condition. the implementation suggested in FreeRTOS documentation requires the following changes:

1. You must use the file FreeRTOSConfig.h modified for the MPLAB X PIC32 environment. Modify to be:

```
#define configGENERATE_RUN_TIME_STATS 1
```

2. add the lines in task.h immediately following the "#include" statements:

```
#if configGENERATE_RUN_TIME_STATS == 1
    extern volatile unsigned long ulHighFrequencyTimerTicks;
    #endif
```

RD10 – LCD Control with RTOS Statistics

This example passes a counter between two tasks. Each task increments the counter before passing it back. If BTN1 is pressed, a message is set to the LCD at the rate of one message per 1/2 second. Since the LCD is set for a 2 second the LCD queue is soon filled. Once the LCD queue is filled, the ping ponging is slowed from the 1/2 second rate to the LCD 2 second rate. When BTN1 is released, the LCD continues to update the display until the LCD queue is empty. When BTN is pressed, the statics are sent to the serial terminal using a serial Tx queue one line at a time. See notes for RD9.

Application Software Architecture

Figure 8 is the data flow model for the stepper motor application. The value of this model is that it assists the developer to partition the problem in to single task operations and establish the required interfaces. Sequence and timing are not an element of this diagram; only the operations and their relationship to other operations. Adding variables names that that are communicated to the diagram allows the developer to write code that implements the function structure.

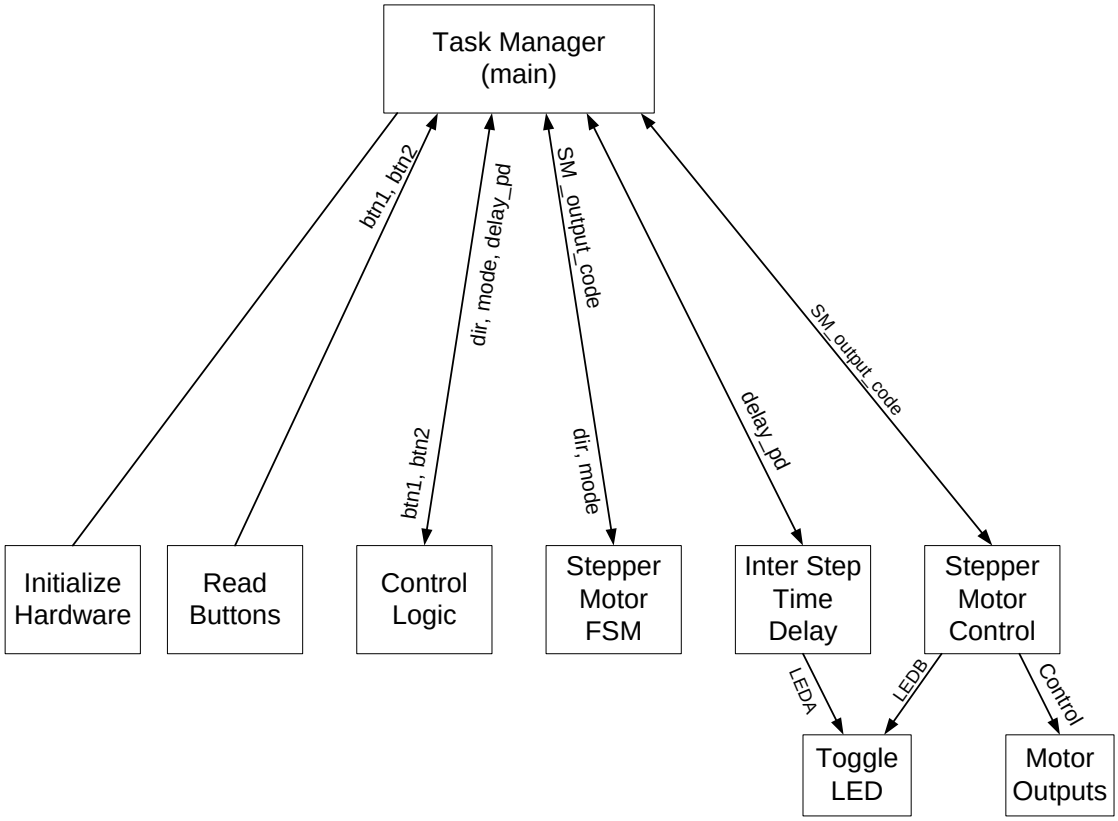

Figure 8. Stepper Motor software model - Data Flow Diagram

Figure 9 shows the control flow diagram for this tutorial. Control flow diagrams describe the order in which tasks or operations need to be completed. Control flow diagrams can use hierarchy to allow graphical representations of control flow at the detail level needed to facilitate understanding. For simplicity, only four graphical elements are needed for control flow diagrams: arrows that indicate program execution flow, the box that represents a process, a diamond that represents decisions, and circles that allow program paths to be joined. The rules are simple: arrows go between boxes, circles and diamonds, boxes have single inputs and single outputs, circles join program paths and can have multiple inputs but only one output, and diamonds have single inputs but two or three outputs. Diamonds show how decisions are made to choose which one of multiple paths is to be selected. Diamonds always ask a question and the outputs represent the possible answers. The answers are either TRUE or FALSE or GREATER THAN, EQUAL TO, or LESS THAN. The path to any process is to be uniquely deterministic. Each one of the process blocks and be further modeled that provides greater detail of how the process is implemented.

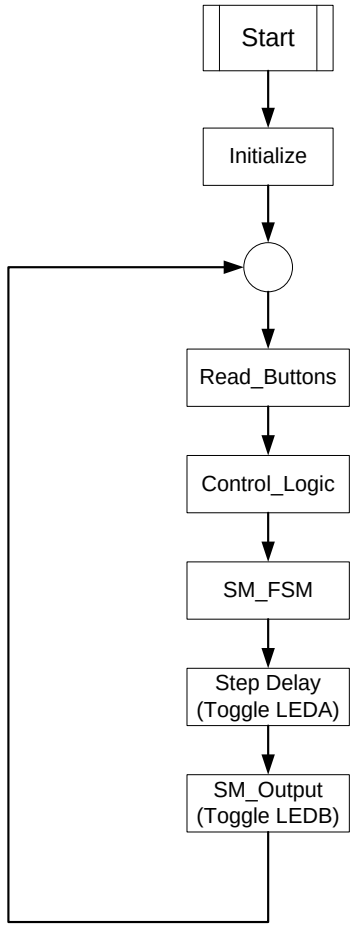

Figure 9. Control Flow Diagram for the stepper motor control using polling

For example, the stepper motor finite state machine process (SM_FSM) can be further modeled using a state diagram as shown in Figure 10. Each state has a specific output code that control the amplitude and polarity of the voltage applied to one or more of the motor coils.[2,3] The parameters shown in the brackets are the conditions required for a given transition that is triggered by a time event.

[2] http://www.freescale.com/webapp/sps/site/overview.jsp?code=WBT_MOTORSTEPTUT_WP
[3] http://en.wikipedia.org/wiki/Stepper_motor

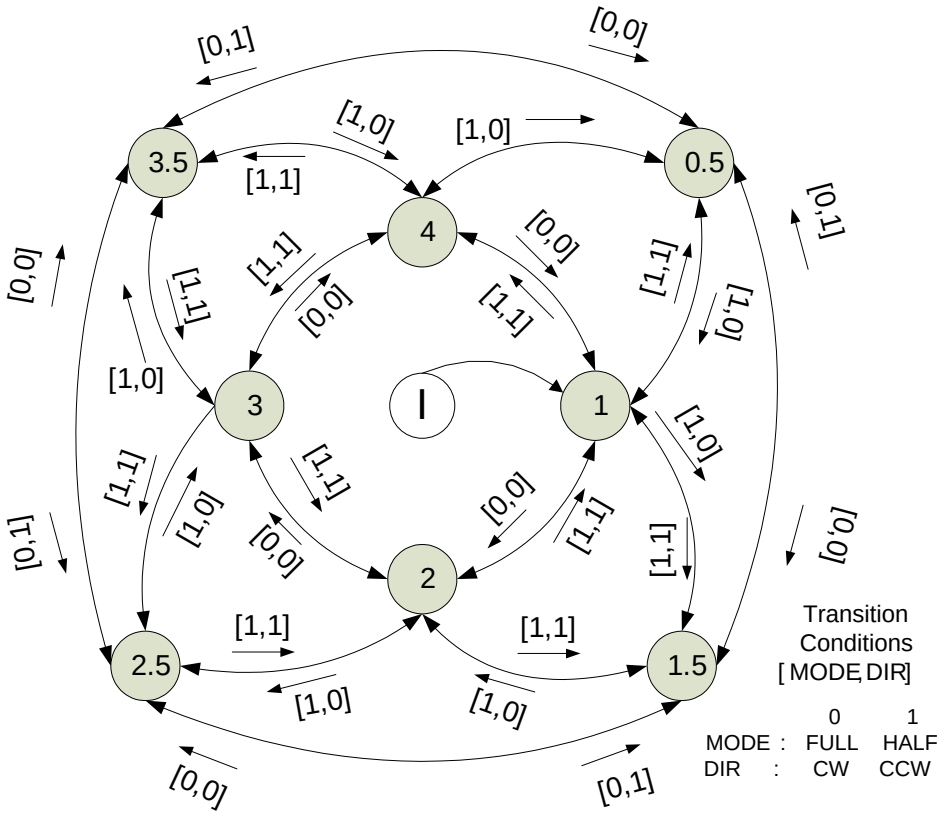

Figure 10. Stepper Motor control FSM

Hardware – Stepper Motor Control

The example provided with this tutorial targets the PIC32 processor running on the Digilent PIC32 MX7ck microcontroller board.[4] The example demonstrates the speed, direction, and step mode control of stepper motor that is the culmination of the laboratory exercises used in the chipKIT™ Pro MX7 Projects 1,through 7. Those projects introduced topics covering digital IO, hardware and software timers, interrupts, handshaking and LCD interface using the Peripheral Master Port (PMP), serial communications, and finite state machine algorithms. If you are unfamiliar with microcontroller design and programming, it is strongly recommended that Projects 0 through 7 be completed prior to attempting to implement the design using RTOS. The hardware configuration for this tutorial is shown in Figure 21 with the parts list is provided in the appendix. The buttons and LCD provide local control and monitoring while the connection to a serial terminal provide for remote operations.

[4] http://www.digilentinc.com/Products/Detail.cfm?NavPath=2,396,986&Prod=CEREBOT-MX7CK

APPENDIX A - FreeRTOS Development for the ChipKIT™ Pro MX7

This appendix describes a method for creating the directory structure for integrating FreeRTOS with a given application. Once the directory structure is set up, new applications can be developed with minimal effort. Although the source code is provided for all FreeRTOS library of files, it is recommended that the user not modify any of the FreeRTOS code. The FreeRTOS source code is compiled at some point in the project build process. The initial build of a FreeRTOS project may take a bit longer than subsequent project builds.

A. The MPLAB X Directory Structure

The FreeRTOS port is the software support can be downloaded off the web at http://www.freertos.org/. Figure 11 shows the directory structure that supports the development using the Microchip MPLAB X 1.8x development environment. New projects are added at the level show for PROJ1 and PROJ2.

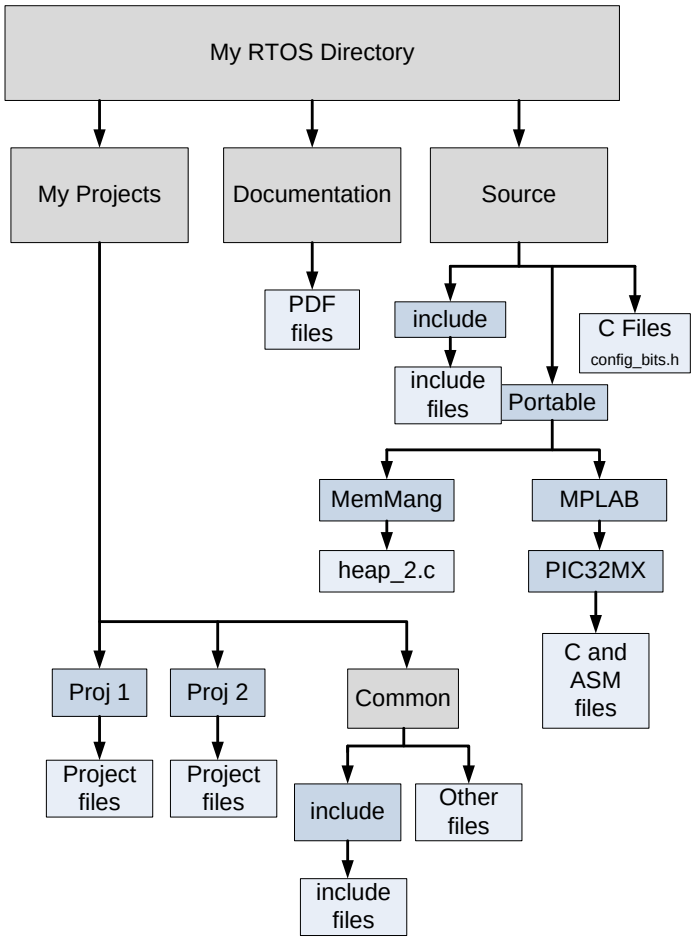

Figure 11. FreeRTOS project development directory structure

B. Steps to create a new FreeRTOS project

By far, the easiest way to generate a FreeRTOS project is to use the Import Legacy MPLAB project wizard. Another way is to save one of the reference designs to a new project folder. Generating an MPLAB X project from

Step I: Create a new MPLAB X project at the directory level shown for Proj 1 or Proj 2 in Figure 11.

Step II: Copy into *FreeRTOSConfig.h*, *config_bits.h*, *chipKIT_Pro_MX7.h*, and *chipKIT_Pro_MX7.c* files into the project directory.

Step III: Add *FreeRTOSConfig.h , config_bits.h* and *chipKIT_Pro_MX7.h* to the project Header Files

Step IV: Add *chipKIT_Pro_MX7.c* to the project Source Files.

Step V: Create a new logical folder under Source Files – *FreeRTOS*.

Step VI: Add all C files in the **My_RTOS_Directory/*Source*** directory.

Step VII: Add *heap_2.c* from the **My_RTOS_Directory/*Source/MemMang*** directory. (*croutine.c, list.c, queue.c, task.c, and timers.c*)

Step VIII: Add *port.c* and *port_asm.asm* from the ***Source/MPLAB/PIC32*** directory.

Step IX: Create a new file called main.c under Source Files. If required, create a new file called main.h under Headers Files.

Step X: Verify that the project window appears as shown in the following figure.

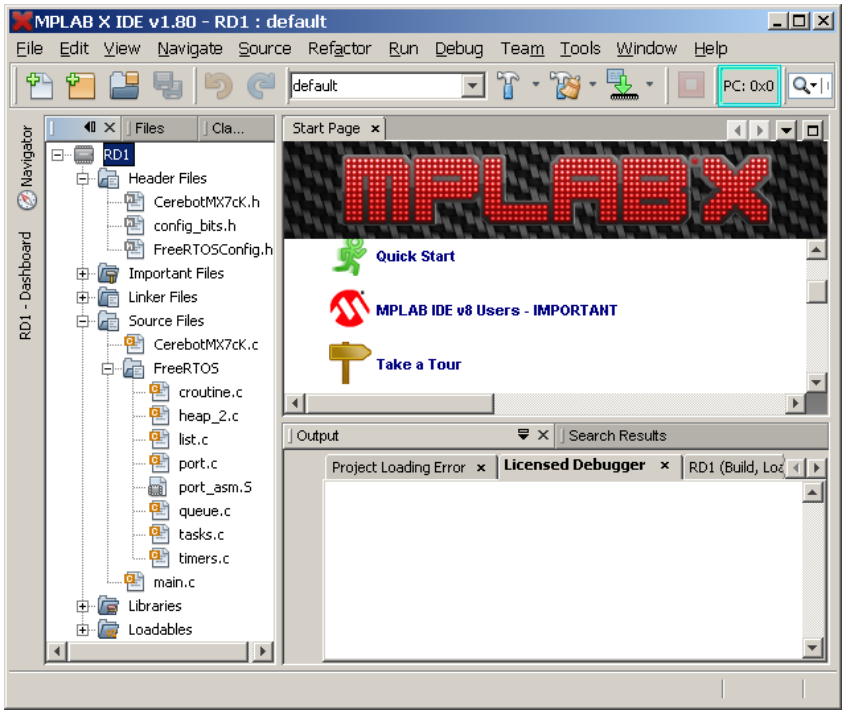

Figure 12. FreeRTOS directory structure

Step XI: Highlight the project Name as shown in Figure 12. Right click the mouse button and select Properties from the bottom of the drop down list. This will result in displaying the project options as shown in Figure 13 based on the selections made during the creation of the project.

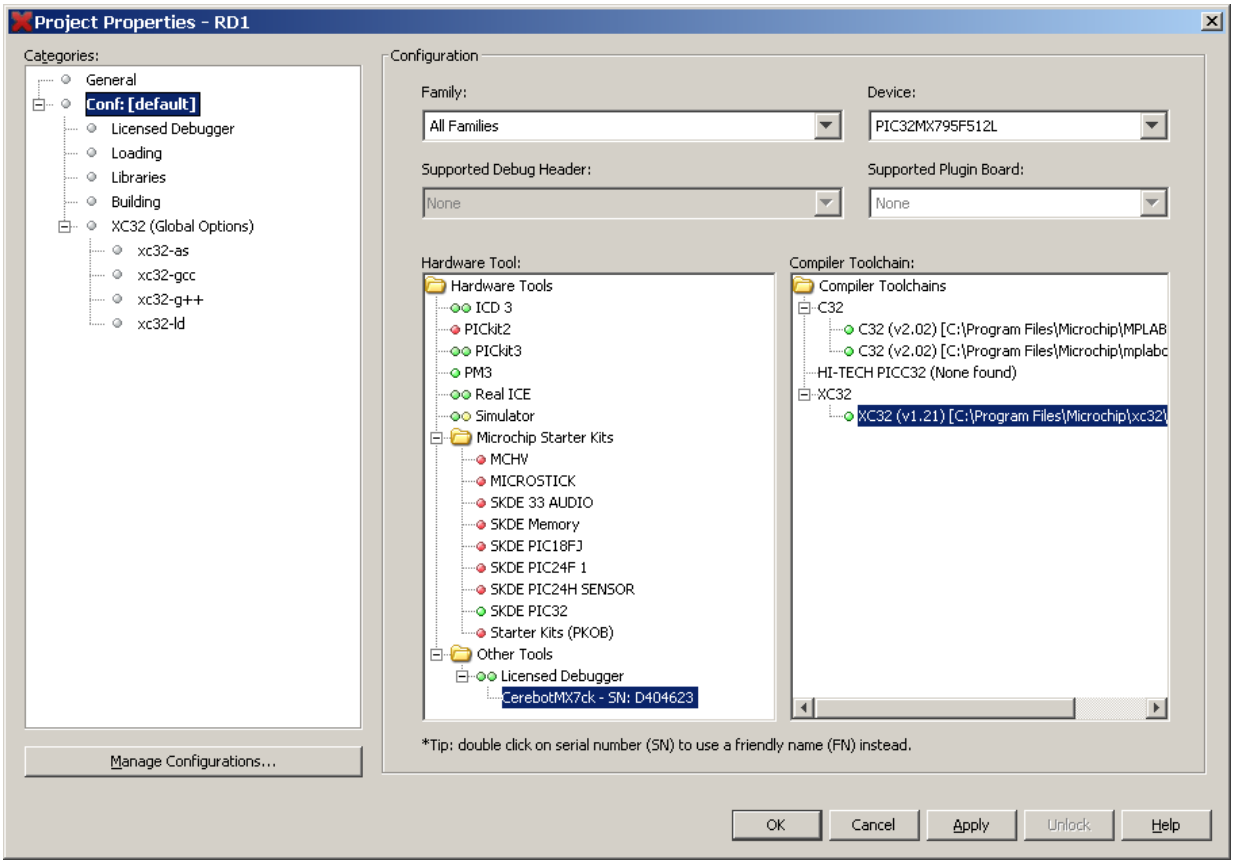

Figure 13. Project Properties window.

Step XII: The following steps modify the project Make File for the XC32 assembler.

Step XII a: Click on the bulleted item labeled xc32-as. A window as shown in Figure 14 is now displayed.

Step XII b. Click on the […] box beside the line Preprocessor Include directories. Enter the text as shown in Figure 15.

Step XII c. Click on the […] box beside the line Assembler Include directories. Enter the text as shown in Figure 16.

Step XII d. Add the text for the entry box for **Advanced options:** *-gdwarf-2*

Step XII e. Click on the box labeled Generate Command Line. The text in the display window should read: *--gdwarf-2 -I"../../common/include" -I"../../../Source/include" -I"../../../Source/portable/mplab/pic32mx" -I".." -I"../../../Source/portable/mplab/pic32mx",-I"..* as shown in Figure 17.

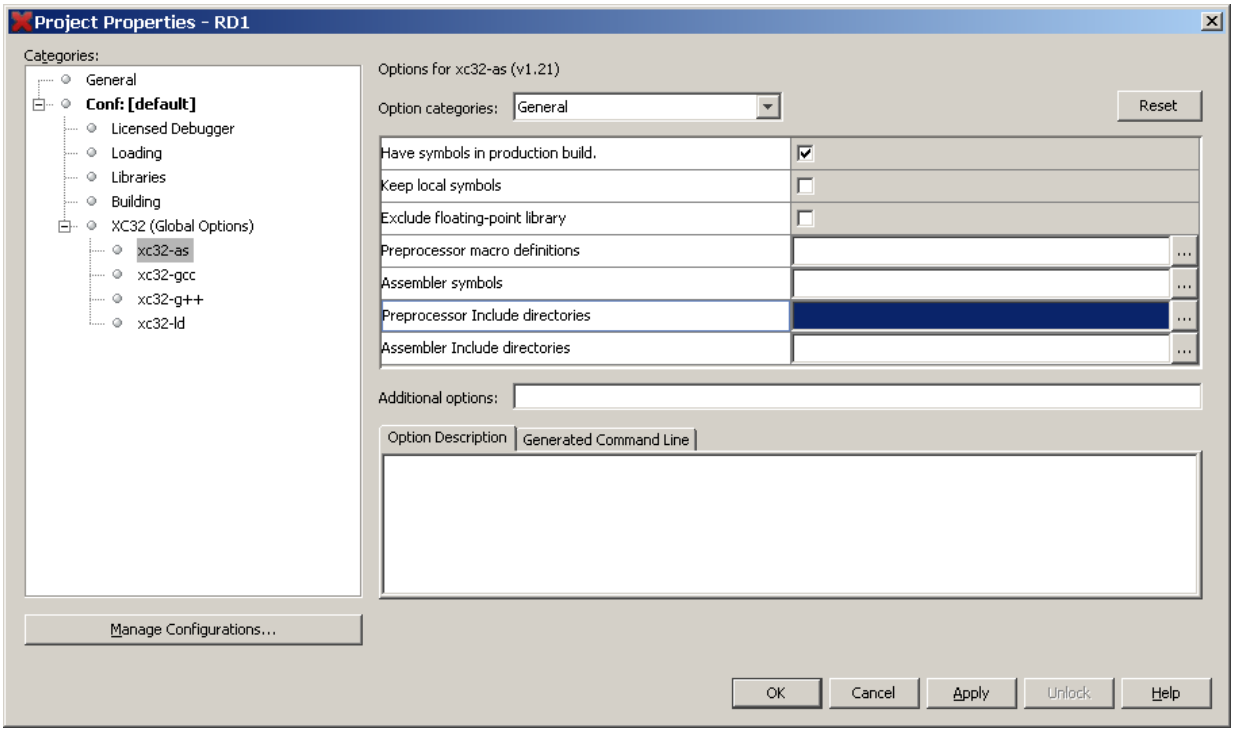

Figure 14. Project Properties XC32-as configuration window

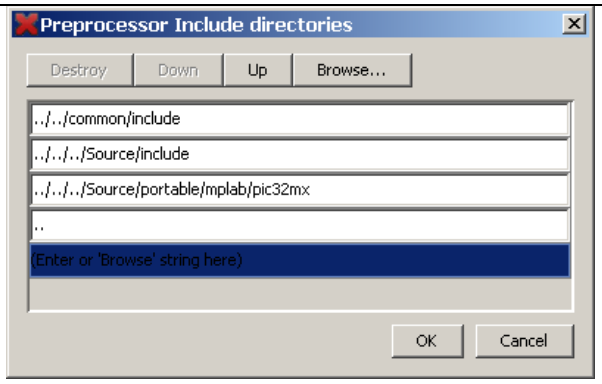

Figure 15. Assembler preprocessor include directory window

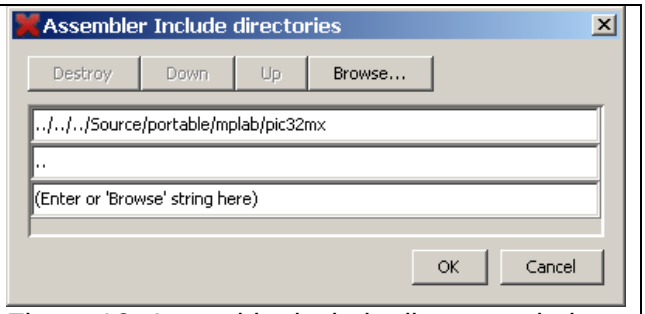

Figure 16. Assembler include directory window

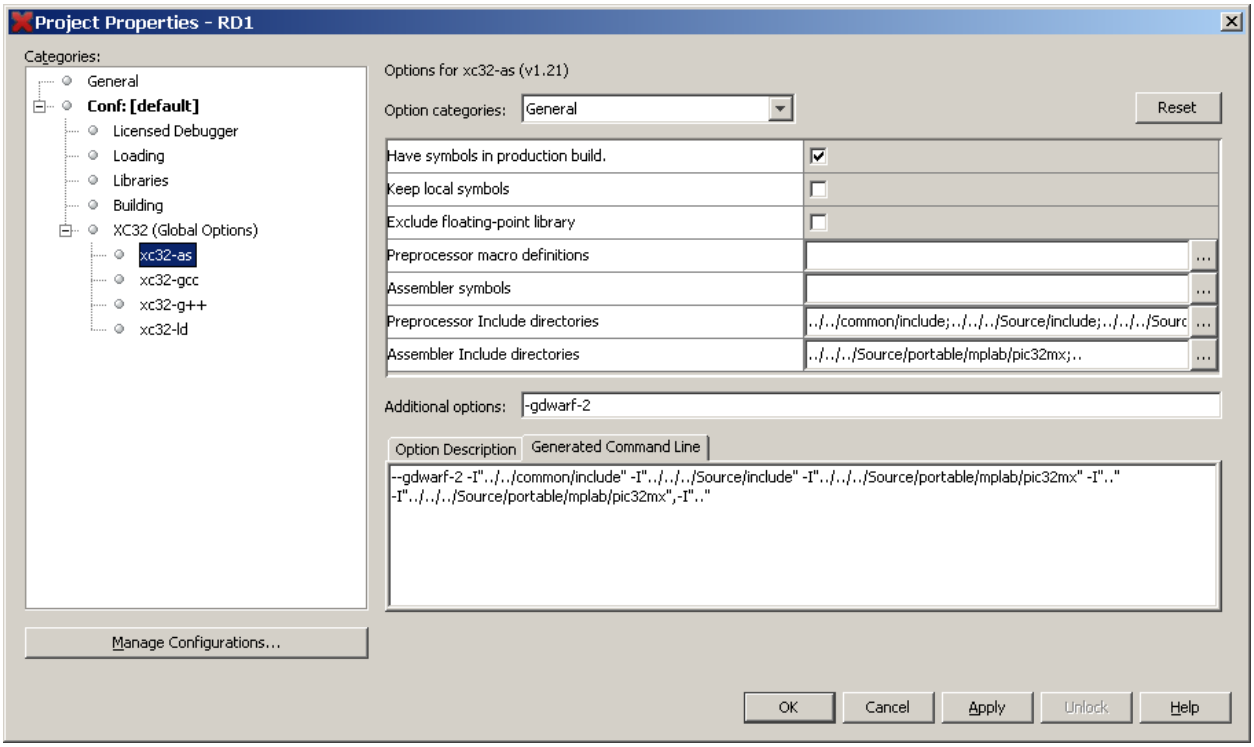

Figure 17. Completed Project Properties window for xc32-as after configuration

Step XIII: The following steps modify the project Make File for the XC32 C compiler.

Step XIII a: Click on the bulleted item labeled xc32-gcc shown in Figure 13. The window shown in Figure 18 will now be displayed.

Step XII c. Click on the [...] box beside the line Include directories. Enter the text as shown in Figure 19.

Step XII d. After completing the Include directories window, click on Generated Command Line. The text in the display box should read: *-g -I"../../common/include" -I"../../../source/include" -I"../../../source/portable/mplab/pic32mx" -I".."* as shown in Figure 20.

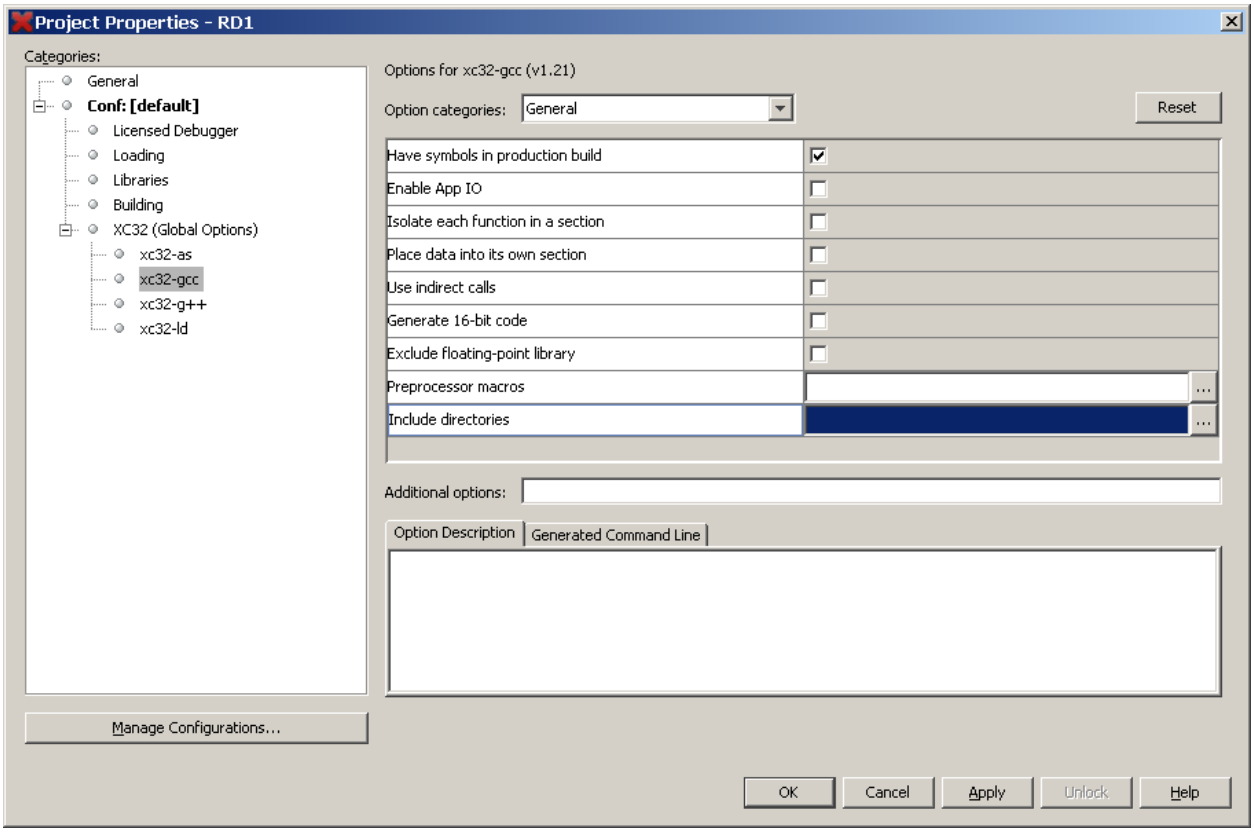

Figure 18. Project Properties XC32-gcc configuration window

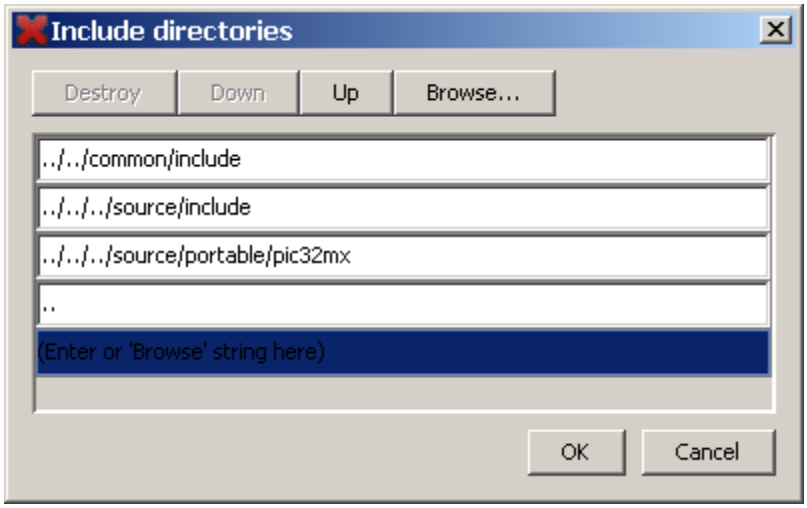

Figure 19. C compiler include directory window

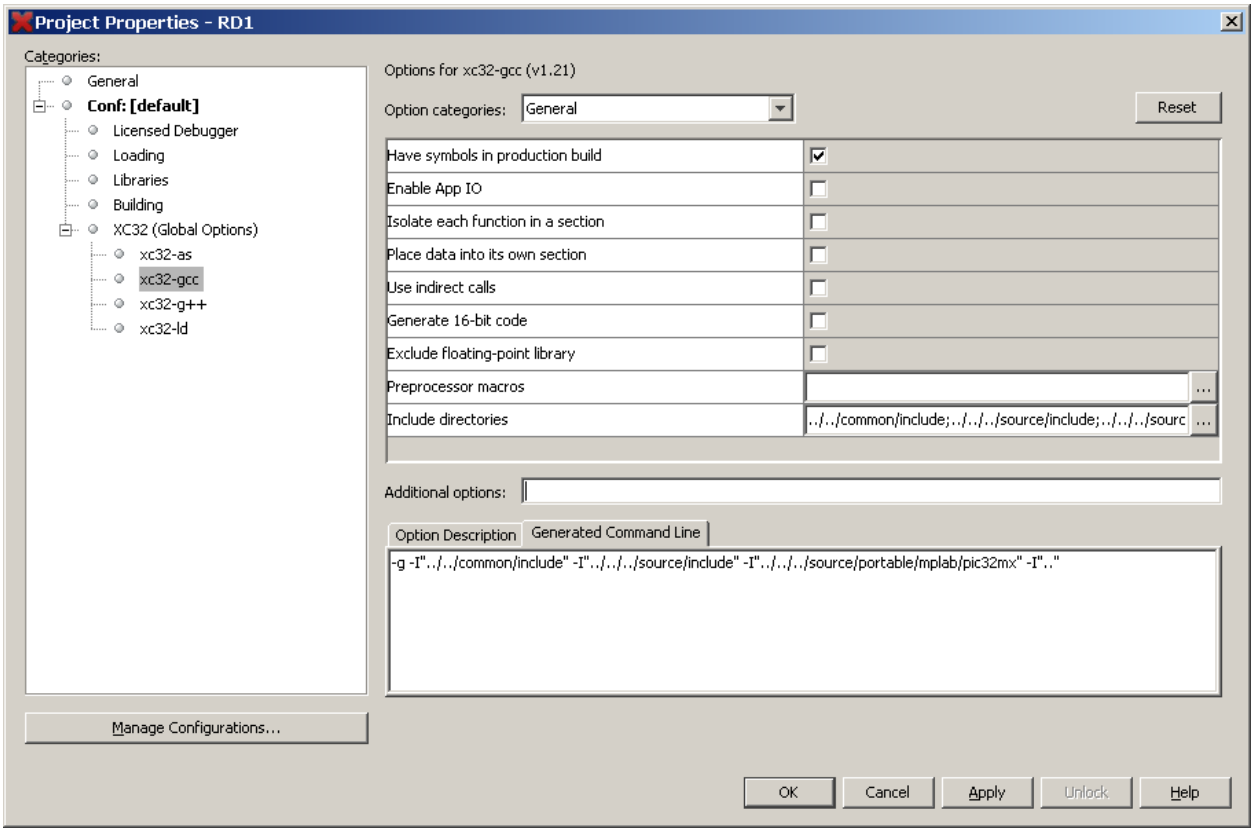

Figure 20. Completed Project Properties window for xc32-as after configuration

APPENDIX B – Hardware Platform for FreeRTOS Reference Designs

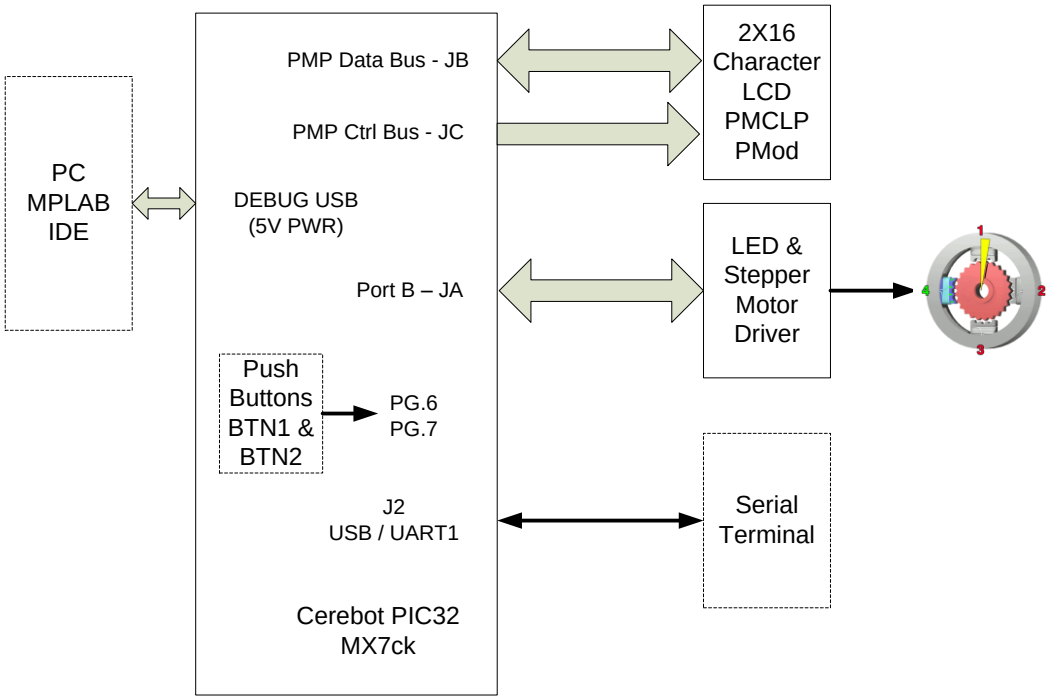

Figure 21. Stepper Motor control hardware block diagram

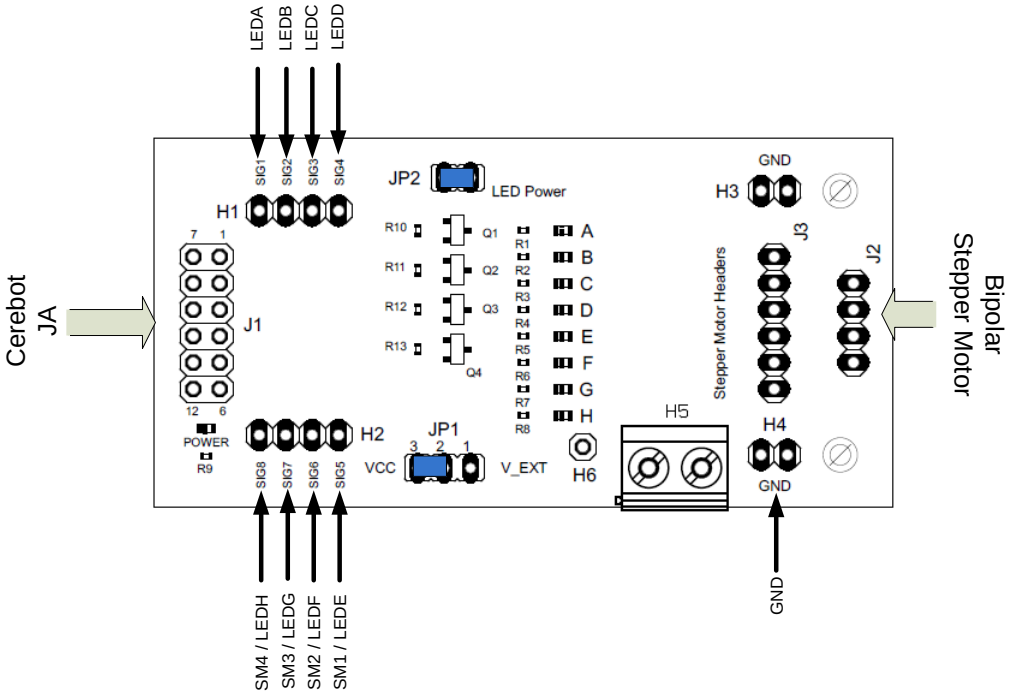

Figure 22. Stepper Motor Driver Module parts layout