Microchip PIC32 FreeRTOS Reference Designs

Dr. Richard Wall – Professor
Department of Electrical and Computer Engineering
University of Idaho
Moscow, ID 83844-1023
Revised December 22, 2012
rwall@uidaho.edu

I. FreeRTOS Reference Designs:

A. RTOS General Notes:

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 management program that allocates the processors resources such that the system performance meets specific timing requirements without conflicts between independent tasks.

The popularity of Free RTOS as an off the shelf (OTS) RTOS[1] is first and foremost it being free. Although it does not have all of the bells and whistles of a full featured RTOS, it provides methods for multiple threads or tasks, queues, mutexes, semaphores and software timers. FreeRTOS currently supports 33 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. A cooperative method depends on a task relinquishing their control of the processor. Cooperative multitasking generally works best when one or more of the tasks are 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.

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

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

B. RTOS Development Environment

1. Hardware – Stepper Motor Control

The example provided with this tutorial targets the PIC32 processor running on the Digilent PIC32 MX7ck microcontroller board.[2] The example demonstrates the speed, direction, and step mode control of stepper motor that is the culmination of the laboratory exercises used in the University of Idaho ECE 341.[3] Those laboratory exercises introduce 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 Labs 0 through 7 be completed prior to attempting to implement the design using RTOS. The hardware configuration for this tutorial is shown in Figure 1 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.

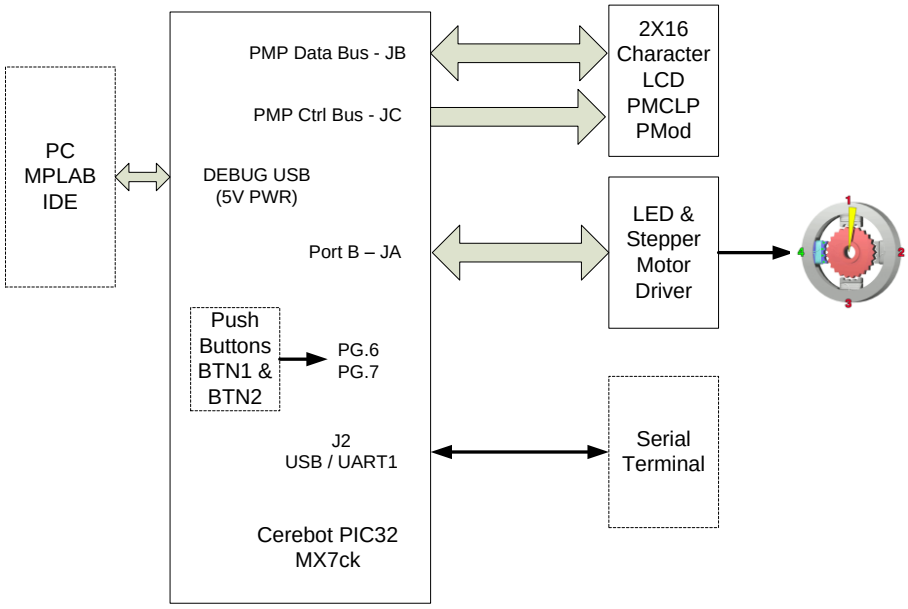

Figure 1. Stepper Motor control hardware block diagram

2. Software Model – Stepper Motor COntrol

a. Application Architecture

i. Figure 2 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

[2] http://www.digilentinc.com/Products/Detail.cfm?NavPath=2,396,986&Prod=CEREBOT-MX7CK
[3] http://www.ece.uidaho.edu/ee/classes/ECE341/labs/labs.html

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.

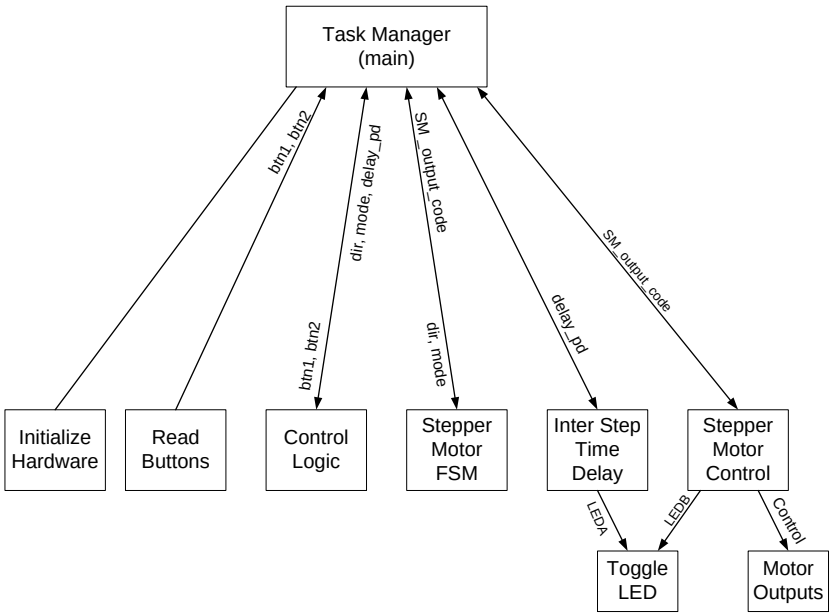

Figure 2. Stepper Motor software model - Data Flow Diagram

ii. Figure 3 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.

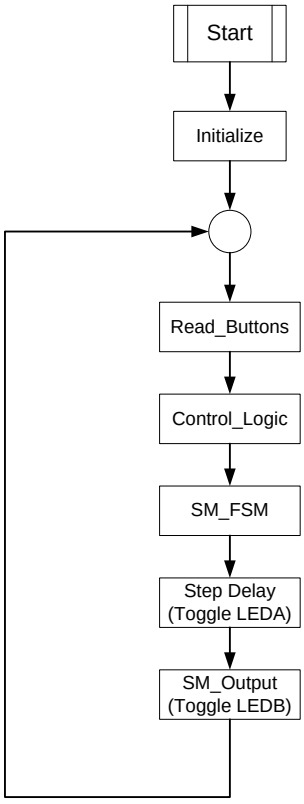

Figure 3. Control Flow Diagram for the stepper motor control using polling

For example, the stepper motor finite state machine process (SM_FSM) can be further modeled using a state diagram as shown in Figure 4. 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.[4,5] The parameters shown in the brackets are the conditions required for a given transition that is triggered by a time event.

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

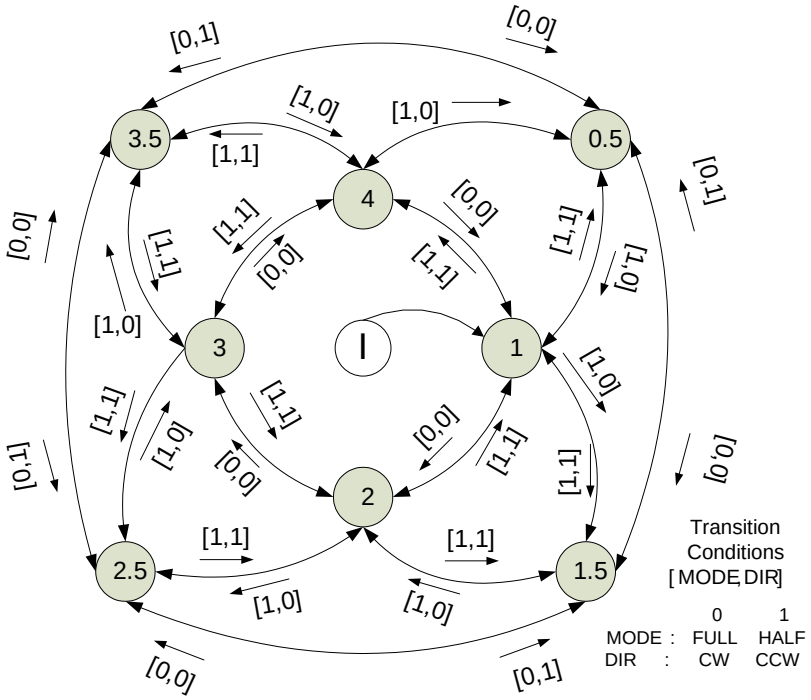

Figure 4. Stepper Motor control FSM

iii. FreeRTOS Development

The FreeRTOS port is the software support can be downloaded off the web at http://www.freertos.org/a00104.html?1. Figure 5 shows the directory structure that supports the development using the Microchip MPLAB 8.86 development environment. New projects are added at the level show for PROJ1 and PROJ2. Figure 6 is the MPLAB project window for this reference design.

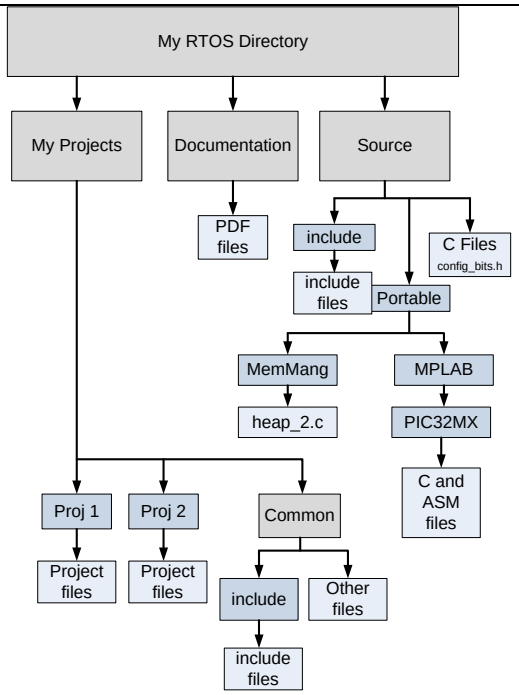

Figure 5. FreeRTOS project development directory structure

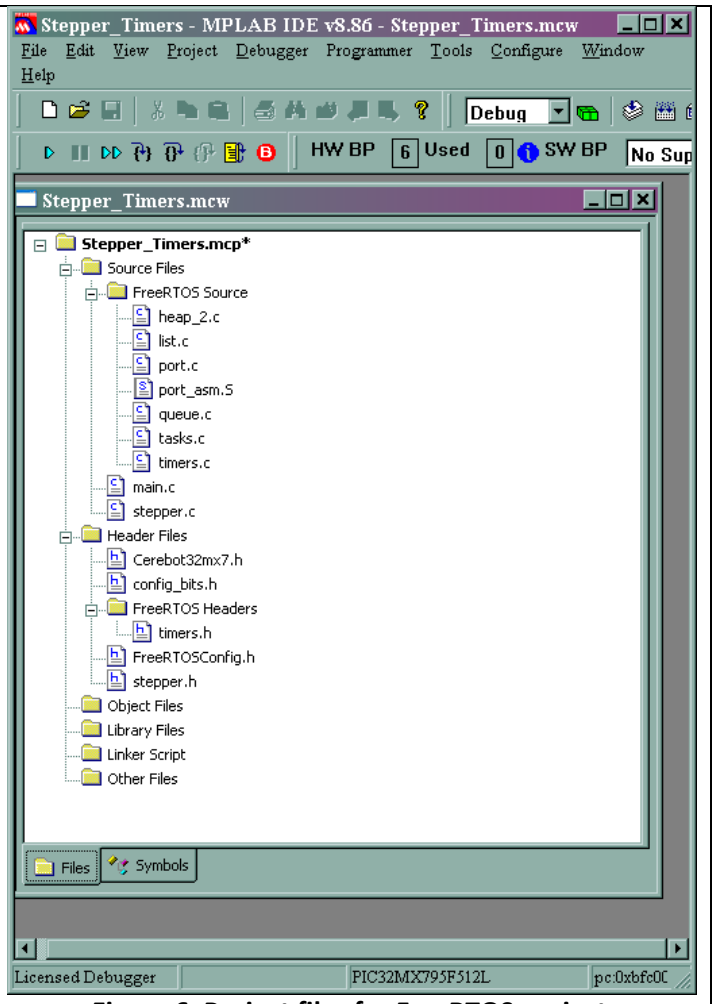

Figure 6. Project files for FreeRTOS project

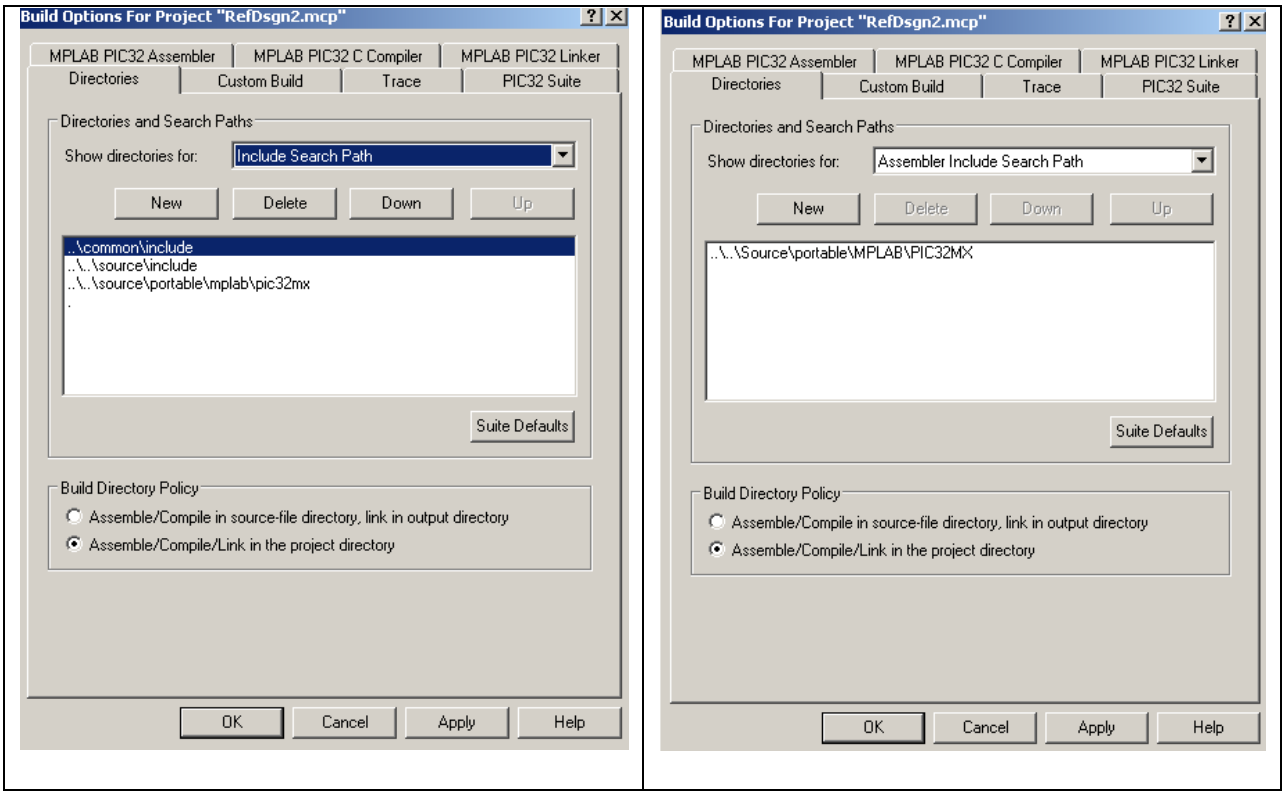

Figure 7. Project settings for the FreeRTOS stepper motor control reference design

C. RTOS Reference Designs

1. RD1 - An example of FreeRTOS running on the Cerebot 32MX with PIC32MX7 processor. The operating system creates manages two tasks that turn on an LED and increments a counter. LEDA is turned on in task1 and turned off in task2. LEDB is turned off in task1 and turned on in task2. A common counter is incremented each time a task turns on the associated LED.

2. RD2 - An example of FreeRTOS running on an the Cerebot 32MX running the PIC32MX7 processor. 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.

3. RD3 - The operating system creates manages multiple tasks that turn on an LED and increments a counter. Four tasks toggling LEDs at various rates. Task 4 is set for a higher priority level than the other three. Both time delay and delay until functions block the tasks for various periods. The operation uses the idle hook task to determine when the scheduler has idle time. The configUSE_TICK_HOOK bit in FreeRTOSconfig.h must be set.

4. RD4 - The operating system creates manages two tasks that toggles a LED and based upon which button is pressed. The program uses a queue to indicate which led to toggle. BTN1 toggles LED1 and BRN2 toggles LED2. The LED task has a 250ms delay while the button has only a 100ms delay. This means that the queue fills faster than the messages are taken out of it. After releasing the button, the LED continues to flash until the queue is emptied. The single prvbutton task serves two button different tasks. The operation of each task is determined by the argument values in pvParameters. Pressing both buttons simultaneously causes the queue to fill twice as fast.

   The operation uses the idle hook task to determine when the scheduler has idle time. The delay in the Note: This program DOES NOT perform a "press on - press off" operation. This functionality is left to a student as an assignment.

5. RD5 - This program controls a stepper motor based upon the status of BTN1 and BTN2. The Button task detects 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. The stepper motor control task is blocked for a semaphore given from the timer 3 ISR. It also checks the queue for messages from the button detection task.

   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.

   The warning generated during the project build operation “Warning: Quoted section flags are deprecated, use attributes instead.” has not been resolved.

6. RD6 LCD_Mutex –This example passes a counter 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 mutex semaphore. Commenting out the semaphore take and give demonstrates how the LCD text gets messed up without using the device exclusion protection provided by the mutex.

7. RD6a I2C_ EEPROM The purpose of this code is to program a I2Cwith 1024 bytes of randomly generated data starting at a random address and verify that the data was correctly saved. See EEPROM_TEST.txt for functional specification details. 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.

8. RD7 UART1 – 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.

Note: You will not see anything on the terminal screen until you press the enter key unless you have the terminal setup for local echo characters.

9. RD8 – TIMERS is an example of FreeRTOS running on an Cerebot MX7ck using the PIC32MX7 processor. 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

10. RD8a – An example of FreeRTOS running on an Cerebot MX7ck 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.

11. RD9 – P1-STATS 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. 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. do not enter the following line in FreeRTOSConfig.h.
        ```
        extern volatile unsigned long ulHighFrequencyTimerTicks;
        ```
    2. add the lines in tash.h
        ```
        #if configGENERATE_RUN_TIME_STATS == 1
            extern volatile unsigned long ulHighFrequencyTimerTicks;
        #endif
        ```

    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.

    Ignore Assembler message warnings concerning quoted section flags

12. RD10 LCD_STATS – 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.