CPE 490 Lab 9 S14

## Goals

1. To get an appreciation of what a Real Time Operating System (RTOS) does in an embedded design.
2. To understand how to construct a kernel that allows real time performance for 2 tasks using just one clock as the source of the real time performance.

## Overview

This lab will require effort and time. Because of this it will count as a lab grade and as the next homework assignment. The ISR, and kernel code are given to you to make this a manageable lab / homework assignment. Make sure that all files needed to build your code are in one directory; that might mean having to copy header files and linker scripts all into the one file. Zip the file along with the written report (including a section for the questions and answers), and send it to me through blackboard under homework assignment #3.

## Design

### Program Overview

Create a program that has three tasks. The tasks will be functions that are called by a kernel. **Two of the tasks will return an integer** that **will specify how many ‘ticks’ should elapse** before the task is called again. The kernel will take care of ‘waking’ up the task after the time has expired.

**Timer1** is used to make the clock tick with an interrupt. The timer is setup so that it counts every 62.5 ns, the PR1 comparator register is set to 1600-1. This will ensure that an interrupt occurs every 1600\* 62.5 ns, or every **100 us**. The ISR, also given to you, simply increments a counter, called **TickCounter.** From this one clock the kernel can provide scheduling for tasks.

The first task scans two input switches: **S3 connected to RD6, and S6 connected to RD7**. The inputs will be **scanned every 10 ms until a push button is activated**. The **switch must be released before any new switch scanning can be done**. To **avoid switch bounce** once a button is pressed the state of the input switches **will not be scanned for 100 ms** and then **if released the scanning will resume to every 10 ms**. No need to delay reading when push button is released since the switch contacts should be far from each other. It is not acceptable to wait in a loop for the pushbutton to be released or to use a delay to wait for the .1 seconds to be over. You must use the kernel to wake you up at the proper time, so that other tasks can still run. In general your code should have no delay functions, where you wait for an event to occur. The few nops in the LCD routines are acceptable.

The S3 switch will cause a 16 bit counter to increment and display the count on the LCD. The display top line will be used and it will look like “Count = XXXXX” where XXXXX is the count. The 16 bit counter will roll over to 0. This task will prepare the string that will be displayed on the LCD showing the count.

The S6 switch will cause the real time clock to be displayed. The display will updated as the real time clock increments. The display will look like “ RTC HH:MM:SS.Z” where H is the hours in military time (range of 00 to 23), M is for minutes, S is for seconds, and Z is either 0 or 5 for the half second. The RTC will be displayed until the S3 switch is pressed at which point the count display will return.

The second task is the Real Time Clock (RTC) task. This task will keep track of the time down to the half second. The task will tell the kernel to call it again every 500 ms so that it can keep the current time. This task prepares the string to be printed on the LCD when the RTC is chosen.

The third task is the lowest priority task. It will send the current display string one character at a time to the LCD. **Do not use the puts\_lcd() function in your code.** In previous labs when we wrote a character to the LCD we would call the check\_busy() routine until the LCD was not busy and then send a character. In order to keep real time commitments we can no longer afford repetitive calls to check\_busy() and then write a character. According the to the data sheet we could be wasting up to 1.6 ms of precious processor time waiting for the LCD to be not busy! So this task will exit if the LCD is busy letting the other two tasks have the processor time, after all how fast can you read a LCD?

### Assignment Details

You are given, on blackboard, the main.c (kernel C code), and the LCD.c code from the previous lab that is modified and its header file LCD.h can be reused. You must modify LCD.c to not call check\_busy before writing data or a command. You will call it from the WriteLCD() function. You will need to write code for the 3 tasks that the kernel will call to complete this assignment.

The main.c code starts out first by including some files:

//\*\*\*\*\*\*\*\* Include files

#include <p33FJ256GP710A.h>

#include "lcd.h" //contains the LCD routines

#include <string.h>

The lcd.h file contains a prototype for check\_busy() so that it can be called from the function you write that writes to the LCD.

Including string.h is not needed but could come in handy for the functions that you will write. For example you can now use the standard C functions like strcpy().

Next the code defines logic TRUE and FALSE and sets up the configuration bits correctly. Next some prototypes for function are given:

//\*\*\*\*\*\*\*\*\*\*\*\*\* Prototypes \*\*\*\*\*\*\*\*\*

unsigned int ButtonPushCounter(void);

void WriteLCD(void);

unsigned int RTCTimer(void);

These are the functions that you will write and are called by the kernel they will be described in greater detail below. We will us the support given to us by the compiler to do context switching when a function is called. An RTOS like FreeRTOS would not use functions like this instead it would provide code to switch context.

Next are some global variables that are declared:

//\*\*\*\*\*\*\*\*\*\*\* Global Variables

char RTCFlag;

unsigned int TickCounter;

char LCDDisplay[16];

RCTFlag will be set true in the ButtonPushCounter() function when the S6 is pressed, RTCTimer() will read this flag and then know to write the LCDDisplay string for later display.

TickCounter is the counter that gets incremented every 100 us as a clock tick in the Timer1 ISR.

LCDDisplay is an array of 16 characters plus a null character used as a string that will be sent to the LCD. Either ButtonPushCounter() or RTCTimer can write this array to be displayed one character at a time using WriteLCD().

We are using global variables to pass flags and data between tasks, in a RTOS this is usually done with a queue.

Next in the program comes the ISR, it is very simple; increment the counter and turn off the interrupt flag.

Next comes the main() function, the first part is used to initialize the PLL for the oscillator, so that the oscillator frequency is 32 MHz and the instruction frequency is 16 MHz.

Next the LCD is initialized and reset to the home position, which is the left, most top character.

Next Timer1 is setup to generate an interrupt every 100 us. This is the resolution for all time keeping functions.

Next the timer1 interrupt is turned on.

Next local variables are declared:

unsigned int bpInitialTick=0, rtcInitialTick=0, Temp, bpWaitCount=0, rtcWaitCount=0;

Both functions, ButtonPushCounter() and RTCTimer() when called will return a value of how many clock ticks or counts of TickCounter must pass before that function should be called again. These values are stored in bpWaitCount or rtcWaitCount respectively.

In order for the kernel to know when to call the function again it will record the current TickCounter value in bpInitialTick immediately after the function ButtonPushCounter() returns, or the value of TickCounter will be written into rtcInitialTick immediately after RTCTimer() returns. The current TickCounter value will then be subtracted by bpInitialTick, and rtcInitial to see how long it has been since the functions have run. This difference is compared to bpWaitCount or rtcWaitCount to see if the function should be called again.

Next is the kernel code:

while (TRUE)//start of the kernal that should never be exited

{

//Since button push has higher priority we will take care of it first.

if (bpInitialTick <= TickCounter) Temp = TickCounter - bpInitialTick;

else Temp = (65535-bpInitialTick)+ 1 + TickCounter;

if (Temp >= bpWaitCount)

{

bpWaitCount = ButtonPushCounter();

bpInitialTick = TickCounter;

}

//Now check to see if the RTC needs to run (it runs every .5 second)

if (rtcInitialTick <= TickCounter) Temp = TickCounter - rtcInitialTick;

else Temp = (65535 - rtcInitialTick)+ 1 + TickCounter;

if (Temp >= rtcWaitCount)

{

rtcWaitCount = RTCTimer();

rtcInitialTick = TickCounter;

}

//This last part is the lowest priority and it prints to the LCD display

//Needs to be written so that if inteface is busy then return but don't

//loose your place in writing the LCD

WriteLCD();

} // end of main program loop

The kernel first checks to see if it is time to call ButtonPushCounter(). This code takes care of the problem when TickCounter rolls over. It looks at the elapsed number of ‘Ticks’ from the last time the function retuned and compare this to the number of ticks the function has requested to wait. If they equal than the function is called.

For this lab you will write the three functions called by the kernel; each is now described.

**unsigned int ButtonPushCounter(void)**

This function will scan both switches every 10 ms. If the S3 is pressed it will increment a counter and write LCDDisplay[16] to display “Count = XXXXX the number of times the push button has been activated. If the S6 is pressed it will set global variable RTCFlag to true to notify RTCTimer that it should be writing LCDDisplay. The function must not retrigger a new reading of the switch until switch is read to be not pushed. The function CAN NOT wait to see the push button go inactive. If the function does not see any active switches then it will return a value to the kernel that will have the kernel call it again in 10 ms. If the a switch is active it will return a value to the kernel to call it again in 100 ms so switch bouncing Is not read.

**unsigned int RTCTimer(void)**

This function will increment clock counters so that an accurate time in hours (military time), minutes, seconds, and half second is kept. If RTCFlag is true than this routine must write LCDDisplay[16] with the correct string. The display should look like “RTC HH:MM:SS.Z” where Z is a 0 or a 5. This function should return an integer so that it will be called again in .5 seconds.

**void WriteLCD(void)**

This function is an idle task; it runs when nothing else is running. The task will write to the LCD display. To make sure that this task is short as possible it will only write one character every time it is called. To keep track of where you are at in the string a nonvolatile variable is needed. The C declaration of “static” might come in handy. In order to not write to the LCD when it is busy the check\_busy function in LCD.c should be used. If the LCD is busy don’t wait for it, return from this function and try again the next time then function is called. In this case it is possible to not write a character every time WriteLCD is called.

## Questions:

1. What would happen if the WriteLCD() function took a long time to complete?
2. What is the maximum time that WriteLCD () can execute before effecting the other tasks?
3. Is our homemade RTOS a preemptive or cooperative scheduler?
4. Will the RTC have an error due to the ISR that will accumulate over time?