Brian Willis

Embedded Cooperative Multitasking Security System

EE 344 – Embedded Microcontrollers II

Dr. Todd Morton

12/11/2017

Introduction

This lab exercise involves the design of an electronic security system utilizing a cooperative multitasking kernel on a Kinetis MK65 Tower System Modular Development Board. This security system alerts the user if specific sensors on the board are triggered via LEDs, an external speaker, and a Liquid Crystal Display (LCD). These triggers include activation of touch-sensing electrodes, ambient temperature outside defined limits, jostling of the board, and system fault detection.

The system has three states: Armed, Disarmed, and Alarm. Certain triggers cause the system to transition from the Armed state to the Alarm state, which outputs an alarm to the external speaker. The system enters the Disarmed and Armed states by user input via the board's keypad.

This report details each task that the security system calls, along with debug bit assignments and a timing analysis of the program.

Program Description

Debug Bit Assignments

Table 1 displays the System Tick Timer, tasks and interrupt service routine used in the kernel with their debugging helper signals used for debugging and timing analysis.

Task	DB #
SysTickWaitEvent()	0
ControlDisplayTask()	1
KeyTask()	2
SensorTask()	3
LEDTask()	4
TermInterfaceTask ()	5
WDOGTask()	6
PIT0_IRQHandler()	7

Table 1 - Final Debug Bit Assignments

Program Module Descriptions

Lab5Main.c

Lab5Main.c is the main module for the program. It initializes the timers and peripherals used by the security system (line 45), and the time-slice scheduler calls its tasks every 10 ms (line 71). The tasks that exist in Lab5Main are ControlDisplayTask() and the LEDTask(). The state of the alarm is an external variable from AlarmWave.c, the state of the electrodes is a global variable from Sensor.h, and the cause of system reset is an external variable from WDOG.c.

ControlDisplayTask() (line 88) has an execution period of 30 ms. This task controls the alarm state and updates the LCD with messages for the user. Its execution period is 30 ms due to the speed at which a user can change the state of the alarm. This task calls functions that detect jostling of the board, representing system tampering, ambient temperature, and user keypresses from the board's keypad. If tampering is detected, "T!" is displayed on the LCD screen and persists until the "C" key is pressed, and if the ambient temperature exceeds its defined thresholds of 0°C and 40°C (line 27), "TEMP ALARM" is displayed on top of the temperature display. If the touch sensors are activated or if the temperature exceeds its thresholds, the system enters the Alarm state if it was in the Armed state. Temperature is displayed in either Celsius or Fahrenheit, and is toggled with the "B" key. The user can control the alarm

system between the Disarmed and Armed states by the “D” and “A” keys, respectively. The current alarm state is displayed to the LCD screen.

There are two techniques implemented to reduce CPU load in this task. First, the tampering, temperature, and current state messages are only displayed on state transitions by retaining the previous loop’s state and comparing it to the current loop’s state (lines 169, 162, and 175, respectively). Similarly, the temperature display is not refreshed until there is a change of at least ±1°C/F (line 127). This way, identical messages are not written to the LCD every loop. Second, several messages have trailing spaces to overwrite previous messages remnants, eliminating the need for clearing the LCD before displaying a message (e.g., line 179).

Temperature in Celsius is calculated from the equation:

$$T = ADC * 0.002582 - 20.51°C$$

and temperature in Fahrenheit is calculated from the equation:

$$T = (ADC * 0.002582 - 20.51) * \frac{9}{5} + 32°F$$

where *ADC* is the result from the board’s Analog to Digital Converter connected to an external temperature sensor. The program keeps the accuracy in the result while avoiding implementing floating-point numbers by scaling the factors up by 10^6, then the result down by 10^6 (line 31).

LEDTask (line 214) has an execution period of 50 ms. This task controls the flashing of the board’s LEDs on its two electrodes. Its execution period is 50 ms due to the periods the LEDs flash at, which are 100 ms and 500 ms. The LEDs flash when their respective electrodes are touched, with a period of 100 ms when in the Alarm state, and 500 ms when in the Disarmed state. When in the Disarmed state the LEDs only flash as their electrodes are touched (line 272), but in the Alarm state the LEDs flash continuously after a single touch of their electrodes (line 237). Furthermore, if the system is in the Armed state, touching an electrode transitions it to the Alarm state. The periods of the LEDs flashing are set by a counter that increments every 50 ms when the task is active (line 299).

Temp.c

Temp.c handles the scanning of an external temperature sensor via an Analog to Digital Converter, placing the current readings in a variable that can be accessed by other modules.

TempInit() (line 15) is called by Lab5Main.c and initializes ADC0 and PIT1, which control the temperature reading. The temperature is set to be read twice per second, so PIT1 is given a load value of 29,999,999 counts per its 60 MHz operating frequency and triggers the ADC to convert a reading from the external temperature sensor. The temperature sensor input is connected to Pin A28 on the board, so DADP3 is set as the input. TempInit() calibrates ADC0 before reading temperatures to increase its accuracy (line 39).

TempGet() (line 50) is called by ControlDisplayTask() (Lab5Main.c line 109) every 30 ms. However, TempGet() will only return a new value every 500 ms as its return value is only overwritten when ADC0 completes a conversion from the external temperature sensor's analog output.

Sensor.c

Sensor.c handles the scanning of the Touch-Sensing Input (TSI) module on the board, placing the current touch readings in a variable that can be accessed by other modules.

SensorInit() (line 22) is called by Lab5Main.c and initializes the TSI. SensorTask() (line 62) then scans the TSI every 10 ms. Its execution time is 10 ms as each touch-sensing electrode requires ~1 ms to scan. SensorInit() is meant to be called at system startup and sets the baseline levels of the electrodes by scanning their initial values when nothing is touching them (line 44, 54). A minimum touch level is then set to be their initially read values plus a defined offset (Sensor.h line 9). When an electrode is scanned by SensorTask(), the electrode's value must be more than the initial value plus the offset to trigger a press (line 72). This means the offset can be increased to decrease the sensitivity of the electrode. The initial values for the electrodes are stored in a global variable such that the rest of the module can read them. However, an improvement to make is that a static global variable should instead be used, and the touch offset definitions should be in Sensor.c instead of Sensor.h to limit the module's scope. SensorTask() is decomposed into two states which switch every 10 ms (line 70). In its first state, the task reads Electrode 2 then scans Electrode 1. The second state reads Electrode 1 then scans Electrode 2. This staggering of the electrodes provides the TSI enough time to scan without using blocking code.

AlarmWave.c

AlarmWave.c handles sending an alarm signal to an external speaker. It includes a private array of 64 values in Q15 notation (line 11) representing one period of an alarm sinewave. This sinewave, illustrated in Figure 1, has a fundamental frequency of 300 Hz and four harmonics at 600, 900, 1200, and 1500 Hz.

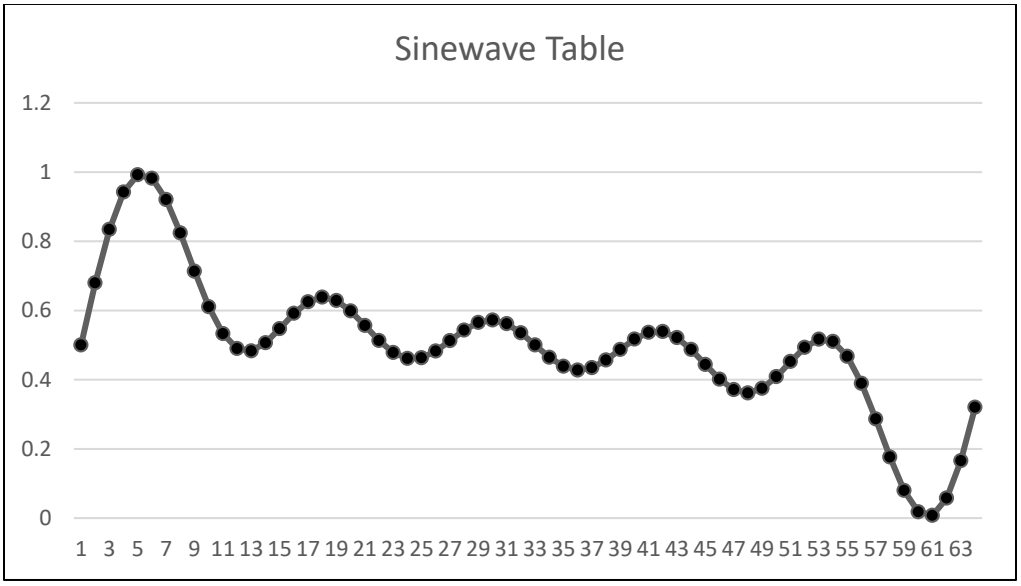

Figure 1 - Alarm Sinewave Generated in Excel

AlarmWaveInit() (line 27) is called by Lab5Main.c and initializes DAC0 and PIT0 that control the alarm output. PIT0_IRQHandler (line 51) is an interrupt that is called every 52.1 µs via a PIT0 load value of 3124 counts (line 37). This interrupt sends one element of the sinewave table to the DAC each time it is called, but only if the system is in the Alarm state. The state of the alarm is a global variable such that both Lab5Main.c and AlarmWave.c can access it (line 20). An improvement to make would be to instead enable and disable the IRQ handler when the alarm is off, which would require less CPU operations.

The frequency of the result from DAC0 is modeled by the equation:

$$F = \frac{1}{52.1\ \mu s} * \frac{1}{64} \approx 300\ Hz$$

WDOG.c

WDOG.c handles the Watchdog timer that detects system fault. Specifically, the Watchdog will detect if the system's total task execution time exceeds the system's timeslice period of 10 ms, which in turn detects blocking code. When the Watchdog is triggered, the system automatically resets and displays a message to the LCD.

WDOGInit() (line 20) is called by Lab5Main.c and initializes the Watchdog. The Watchdog is first given its two unlock codes (line 23), then given its timeout value of just over the timeslice period of 10 ms (line 26). It then checks to see the cause of the system's last reset (line 33). If it was due to a Watchdog timeout, it sets a flag that is read by Lab5Main.c such that "W!" is displayed on the LCD (Lab5Main.c line 65). The message could have been displayed while in WDOG.c, but that would have required that LCD.h be included in the module. WDOGTask() is called every timeslice regardless of its duration. The task refreshes the countdown back to just over the timeslice (line 45), so that it can detect blocking code on the system's next loop.

The timeout value for the Watchdog is calculated with an input frequency of 60 MHz via the equation:

$$WDOGTimeoutPer = \frac{TIMESLICE * 60\ MHz}{1000} + 1$$

where *TIMESLICE* is in ms. Since the timeslice in the system is in units of 100 us instead of ms, WGOGTimeoutPer is divided by 10000 instead of 1000 (line 26). WDOGTimeoutPer for a timeslice of 10 ms will then come out to be just over 10 ms.

SysTickDelay.c

SysTickDelay.c handles the System Tick Timer for use by the timeslice scheduler. The module was written by Dr. Todd Morton, but SysTickWaitEvent() was modified for this lab to calculate peak CPU load. The module has four new global variables that SysTickWaitEvent() sets (SysTickDelay.h line 13), where the current and previous loops' CPU loads and cumulative task execution times are stored.

CPU Load is calculated by the equation:

$$CPU\ Load = \frac{Time\ since\ last\ loop}{TIMESLICE} * 100\%$$

where *Time since last loop* is the number of 100 µs units the scheduler's last loop took and where *TIMESLICE* is in units of 100 µs (line 49). For example, if the previous loop took 400 µs and the timeslice is 10 ms,

$$CPU\ Load = \frac{4 * 100}{100} = 4\%$$

However, the load is set as 10*CPU Load in the function such that the CPU Load can be displayed with a resolution of 0.1% in TermInterface.c. The timeslice was shortened to 100 µs units instead of 1 ms units with the intention of achieving this resolution (line 26), but it was found that timeslice units of 10 µs are actually required to achieve this resolution. Therefore, as the program is, the CPU load will have a resolution of 1.0%, meaning further work is required to achieve this design goal of the CPU load calculator.

Cumulative task execution time is calculated and stored in a separate global variable, and is similarly calculated by the equation

$$Cumulative\ task\ execution\ time = Time\ since\ last\ loop * 100$$

where *Time since last loop* is in units of 100 µs (line 51).

Only the CPU Load and cumulative task execution time peaks are meant to be displayed, so the scheduler's previous loop's CPU Load and cumulative task execution time are stored as global variables (line 59). The CPU Load and cumulative task execution time are only overwritten with new values if greater than the previous values (line 50, 52).

CPU Load and cumulative task execution time peaks are calculated once per timeslice, and do not take into consideration the execution periods of the tasks. This means that the execution periods of the tasks are all taken as 10 ms, and therefore the results will be significantly larger than the results of the Timing Analysis (page 13). For example, the peak cumulative task execution time for the program is ~2.8ms, so the total CPU Load calculated is 28.0% as opposed to the ~10.7% from the Timing Analysis, meaning further work remains to meet this design goal of the CPU load calculator.

BasicIO.c

BasicIO.c handles all aspects of the terminal interface. The module was written by Dr. Todd Morton, but BIOGetStrg() and BIOGetChar() were modified for this lab to read user input to the terminal without implementing blocking code. TermInterfaceTask() in TermInterface.c creates a static array of six characters that is passed into BIOGetStrg() and each element of the array is set to the current input to the terminal if it is a displayable character (line 182). BIOGetChar() reads the current input to the terminal, which will be mostly null due to the speed at which the function is called (line 64). A static variable keeps track of the current character in the string to write to in BIOGetStrg(), and is updated to keep up with the user's keypresses (line 185, 191, 196). A return variable of 1 indicates the user exceeded six characters in their input string or hit enter, causing TermInterfaceTask() in TermInterface.c to accept the inputted string.

TermInterface.c

TermInterface.c handles the terminal interface for displaying CPU load. If the user inputs “load” to the terminal, the current peak CPU load and cumulative task execution time are displayed (line 37), and if the user inputs “reset” the peak CPU load and cumulative task execution time are reset to 0 (line 43). The characters after “load” and “reset” are ignored, so “loadd” for example would still display the results to the terminal.

TermInterfaceTask() is called by Lab5Main.c and has an execution period of 30 ms. Its execution period is 30 ms due to the maximum speed at which a user can press the keys on their keyboard. The CPU load is calculated in SysTickDelay.c, stored in a global variable, and then accessed in TermInterface.c to display to the terminal. BasicIO.c is implemented to write to and read from the terminal, but was modified to be nonblocking and initialize the UART to have a baud rate of 115200 instead of 9600. TermInterfaceTask() is decomposed into states (line 57) such that the execution time of the function does not exceed the timeslice of 10 ms. This eliminated the need to modify BIOPutStrg() and BIOWrite(). With this implementation, only several characters are displayed to the terminal every 30 ms, which is slow enough not to restrict the other tasks but fast enough to display quickly enough to the user’s eyes.

Timing Analysis

Table 2 displays the Timing Analysis for the program, measured via persistence mode on an oscilloscope. All tasks presented are worst case scenarios, and were achieved via combinations of thoroughly diverse user inputs.

Task	Max Execution Time	Execution Period	CPU Load
ControlDisplayTask()	2.75 ms	30 ms	9.12%
KeyTask()	6.4 µs	10 ms	0.064%
SensorTask()	1.5 µs	10 ms	0.015%
LEDTask()	1.7 µs	50 ms	0.0034%
TermDisplayTask()	1.82 µs	30 ms	0.0061%
WDOGTask()	180 ns	10 ms	0.0018%
PIT0_IRQHandler()	776 ns	52.1 µs	1.49%

Table 2 - Timing Analysis for Security System Program

SysTickDelay() was measured to have a max execution time of ~7.17ms. Therefore, the peak cumulative task execution time should be 10ms – 7.17ms = 2.83ms. This is very close the measured peak cumulative task execution time of ~2.76ms.

The max execution time in ControlDisplayTask() comes from changing the alarm state via the keypad and the time it takes to display the corresponding message to the LCD. Writing to an LCD typically is a time-intensive process, as exemplified here.

The CPU loads per task are calculated by the equation

$$CPU\ Load_{Task} = \frac{Task\ Execution\ Time}{Task\ Execution\ Period} * 100\%$$

The max CPU load for the whole program is calculated by summing the CPU Loads per task together.

Max CPU Load ≈ 10.7%

Comments and Conclusions

This lab provided extensive experience in cooperative multitasking. Specifically, this lab built upon and required an understanding of timing such that numerous tasks in a program can complete their functions quickly without impeding one another. The Watchdog and Terminal Interface modules were the greatest examples of this.

The Watchdog was not only a learning tool but a great way to conveniently debug the program for timing issues. There were several times during the development of the program where blocking or severely inefficient code was automatically detected by the Watchdog, which in turn alerted of poor code structure.

The Terminal Interface required an understanding of efficient cooperative multitasking due to the serious timing issues that were present during its development. State decomposition was necessary to create this task, and timing analysis on an oscilloscope was used to ensure it was reasonably efficient. Furthermore, the Terminal Interface was a great learning exercise in that it required analyzing and modifying modules written by someone else, a vital skill for engineers.