ECE 476 LABORATORY 3

SECURITY SYSTEM
WEDNESDAY, MARCH 7, 2007

ADRIAN WONG AW259
BHAVIN ROKAD BKR24

INTRODUCTION AND PROBLEM FORMULATION

The purpose of this lab is to create a security system using a keypad, LEDs, and serial communication. An administrator using HyperTerminal must be able to concurrently modify the security parameters while a user inputs security codes. Both successful and unsuccessful attempts must be logged to the administrator's console.

HOMEWORK QUESTIONS

1. The alternative is to strobe the output of each column. For example, set all the column outputs to low, and then iteratively set each column output high and then low. A depressed key will cause the row input to read high when the corresponding column output is pressed as well. The disadvantage is that a short between power and ground is possible if two buttons in the same row are pressed at the same time.

2. Two methods can be used. The first is to keep three separate variables (days, hours, and minutes). When the correct number of ticks is generated off of your interrupt (e.g. 60000 ticks of 1 ms), the minutes counter should be incremented. When the minutes counter reaches 60, it should be reset to zero, and the hours counter should be incremented. When the hours counter reaches 24, it should be reset to zero, and the days counter should be increment. The second method is to keep a single time variable (in our case, seconds from January 1) and increment it every 1000 ms. Whenever time needs to be input or displayed, the seconds counter can be converted into days, hours, and minutes through either division or modulus operations. For example, minutes can be derived from the seconds variable by evaluating the following: $minutes = (seconds / 60) \% 60$.

3. The crystal used on the ECE 476 protoboard is the MP160 crystal from CTS Reeves. The listed calibration tolerance is +/- 0.005% (+/- 50 ppm).

HIGH LEVEL DESIGN

The security system was designed around several modules, the local user interface, the remote user interface, and a control/interrupt system.

LOCAL USER INTERFACE

The user is able to enter four digit security codes using a keypad interface. The local user interface scans the keypad buttons and debounces the button presses in order to register only one keystroke per button press. Once a code has been completely entered it is verified against the set of eight valid codes. Every code entry is logged to the administrator's console.

REMOTE USER INTERFACE

The remote user interface allows an administrator to modify the security system. Operations include:

- adding access codes
- deleting access codes
- listing all existing access codes
- remotely unlocking the door

- forcing a system lockdown
- releasing a lockdown
- setting the time

The interface is accessed using any serial based text terminal (e.g. HyperTerminal on Windows).

DOOR CONTROL / INTERRUPT SYSTEM

The security system consists of a timer, transmit, and receive interrupts which allow the microcontroller to service hardware events, such as an empty transmit buffer, in a consistent and organized manner. This module also initializes the interrupts and global variables upon reset.

This section also implements the door lock/unlock and set/reset lockdown routines. These routines ensure that door is only opened when there are no lockdowns in progress, the local lockdown is released after one minute, and the door is relocked after being open for three seconds.

IMPLEMENTATION DETAILS

In order to create an adaptable and manageable structure, we dismantled the project into 5 files:

`main.c` contains **`main()`**, initialization routines, and interrupts. Encapsulates other files using #include.

`local_ui.c` manages local keypad input using **`key_scan()`** and **`debouncer()`**. Checks if entered code is valid that entered and logs activity to HyperTerminal.

`remote_ui.c` manages remote input from the administrative console (HyperTerminal).

`door_state.c` manages opening and locking the door. Checks and sets local and administrative lockdown parameters as necessary. Ensures the door is relocked three seconds after opening.

`defs.h` allows for a centralized location to declare variables and parameters that are used across the multiple files.

The sections below describe the specifics regarding the above segments.

MAIN() AND INTERRUPTS

The main function initializes the ports, timers, and other variables. Timer 1 is set to generate interrupts every millisecond by setting **`tick_1ms`**. The USART is set to operate at 9600 baud and to generate interrupts when a received string is complete or the transmission data register is empty. While inside the main while loop, four different conditions are tested on each cycle.

The first test is if the one millisecond tick has been set by the Timer 1 interrupt. For each one millisecond tick that has occurred, the **`tick_1ms`** flag is reset and the **`tick_30ms`** and **`tick_1s`** are both incremented by one. For each 30 milliseconds tick that occurs, the **`keypad_buffer`** for the local UI is called. For each 1000 millisecond tick that occurs, the **`seconds`** time counter is incremented and the **`door_update`** is called.

The second test is if the local UI has any input pending. This is set via the **`lo_ready`** bit by the keypad buffer. This indicates that four digits has been typed into the keypad and is ready for processing by the **`local_ui`**.

The third test is if the remote UI has any input pending. This is set via the **rx_ready** bit by the **USART_RX** interrupt. This indicates that the remote administrator has pressed enter and their command is ready for processing by the **remote_ui**. Also, the receive buffer on the USART is reset to wait for more inputs.

The fourth test is to see if the remote UI needs to print out the main menu again. The main menu cannot be immediately displayed, since there may still be output waiting to be printed to the remote terminal. This fourth condition waits until the transmission port on the USART is ready to go, and it will call the **remote_ui** to display the menu for the remote administrator.

The **USART_RX** interrupt loads a character from the USART data register **UDR** into the software receive buffer **rx_buffer**. If the user has pressed enter, the software receive buffer is terminated with a null character and the **rx_ready** bit is set high. An echo of the received character is also sent back to the remote terminal to allow the administrator see what was typed in.

The **USART_DRE** interrupt loads a character from the software transmit buffer **tx_buffer** into the USART data register **UDR**. This continues until reaching the null character terminator. An additional feature was added to show a "prompt" that gave a quick indication of the current state of the machine. For example, "!" indicates an open door, "@" indicates remote lockdown, "#" indicates a local lockdown, and ">" represents normal operation.

LOCAL USER INTERFACE

The local user interface control consisted of four functions: **local_ui()**, **keypad_buffer()**, **debouncer()**, and **key_scan()**.

The key_scan function is identical to that used in lab2 in which the input pins are read by sequentially scanning the keypad's horizontal and vertical active low control lines. The detected input is then compared against an array of valid inputs in order to determine which key had been pressed. If a valid input was detected a the **valid_key** flag was set and **debouncer()** further processes the input.

A debouncer function ensures that a slow response time of the human finger did not result in the microcontroller interpreting a single keystroke as many button presses. This functionality was implemented via a state machine outlined in Figure 1. When a button entered the Pushed state, the debouncer saved the valid key for further processing in **keypad_buffer()**.

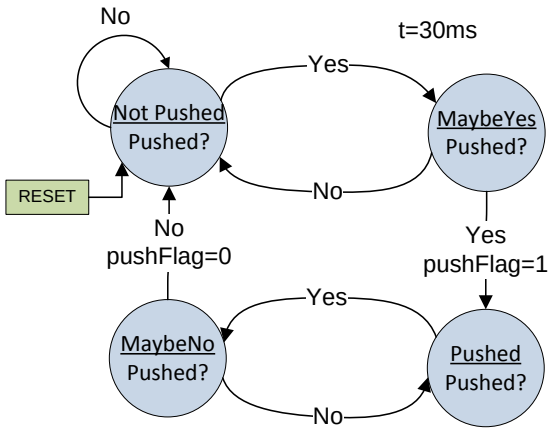

Figure 1: Keypad Debouncer State Machine

The keypad_buffer function sequentially places four individual valid key presses into a character array. It also adds an offset of 0x30 is added to each key input number to convert it to an ASCII value. The ASCII conversion allows **`local_ui()`** to use sscanf to parse and combine the elements of the character array into an unsigned int.

The local_ui function then checks if a valid code was entered by comparing the combined integer against the eight valid codes (stored in an integer array in eeprom). Depending on the code entered the door will be unlocked (by calling a function in door_state.c). The activity will also be logged to HyperTerminal and three unsuccessful attempts results in a one minute lockdown.

REMOTE USER INTERFACE

The remote interface has a five state FSM: **`SHOW_MENU`**, **`MAIN_MENU`**, **`ADD_CODE`**, **`DEL_CODE`**, and **`SET_TIME`**.

The majority of the work is done by the **`MAIN_MENU`** state. The string the software receive buffer **`rx_buffer`** is read in by the FSM. At the main menu, each command is a single character. Thus, the switch statement chooses between all the different possible commands. Some commands (such as List Code, Unlock Remotely, Force Lockdown, and Release Lockdown) can be executed immediately by calling the appropriate function (**`sprintf`**, **`unlock_door`**, **`force_lockdown`**, **`release_lockdown`**). The next state is set to **`SHOW_MENU`**. Other commands, however, require additional input.

Those commands (Add Code, Delete Code, and Set Time) have their own states: **`ADD_CODE`**, **`DEL_CODE`**, and **`SET_TIME`**. This allows them to read in the next string of input from the software receive buffer. **`ADD_CODE`** iterates through the code list until an open slot is found and places the new access code there. **`DEL_CODE`** deletes the access code in the specified slot number. In both cases, an empty slot is set to be "55555", which is outside the bounds of any valid four digit code. Lastly, **`SET_TIME`** takes the users three inputs (days, hours, and minutes) and converts it to the number of seconds since January 1. All three of these states set their next state to **`SHOW_MENU`**.

The **`SHOW_MENU`** state simply displays the default main menu to the administrator and transitions to the **`MAIN_MENU`** state to wait for input.

TESTING AND RESULTS

The primary difficulty we encountered occurred when the administrator attempted to set the time. When setting the time, our remote user interface asked the asked the administrator to enter the time in day, hours, and minutes with a space as a delimiter. The code is question was:

```
sscanf(rx_buffer, "%d %d %d", &days, &hours, &minutes);
```

When the user would enter a valid string such as "1 2 3", the sscanf command above would set days=1 and minutes=3, but would never set the hours variable. We tried numerous solutions such as using #pragma regalloc- to ensure the three variables were stored in RAM. However none of these resolved the issue. We eventually circumvented the bug by changing the data type of days, hours, and minutes to unsigned integers rather than a mix of unsigned integers and unsigned char's.

We tested the security system under a variety of scenarios, constantly modifying the system's parameters from the administrative console to ensure concurrency. Valid and Invalid codes were tested to ensure that they were only accepted when no lockdowns were in progress and that three invalid codes lead to a local lockdown for one

minute. We also ensured consistency if the administrator added or delete codes while a user was entering them. If the administer tried to add a 9th code (there are a maximum of 8 valid codes) they were prompted to remove a code before entering a new code. Listing codes, remote unlock, and remote set/reset lockdown commands were also tested. After the finding the bug in **sscanf**, we extensively tested the system's time keeping and activity logging features. Correct rollover after 59 seconds, 59 minutes, and 23 hours was tested as well.

CLOSING COMMENTS

This lab taught us how to us the auxiliary COM port in conjunction with UDR to communicate with a administrator using HyperTerminal. The concurrent nature of the security system taught us to simultaneously manage multiple functions on the microcontroller and avoid blocking instructions.

The keypad was decoded using the same scheme as cricket call lab (described in the Local User Interface section of Implementation Details). In contrast to lab 2, we added 0x30 to the inputted integer prior to storing them in the character array. This stored the integers in ASCII and allowed us to use **sscanf** to efficiently convert the elements of the character array into a combined integer prior to checking the validity of the entered code.

Figure 2: Circuit Schematic