Introduction

The purpose of this lab was to build a security system with the ability to be administered using a PC and operated using a keyboard concurrently. The administrators of the security system must be able to add 4 digit security codes, delete codes, set the date, force an unlock, force a lockdown, and list all codes as well as any other implemented functionality. We interfaced this through a help menu, which shows what inputs and parameters are needed for each functionality selection. The user should be able to input 4 digit codes and see the result through the appropriate LED lighting up which represents a lock of a door unlocking or locking. If successful LED 0 will light up for 3 seconds and if incorrect LED 7 will light up for 3 seconds. Also, every time the user inputs a code this code is sent to the administrator using the UART interface and indicates whether or not the code was correct. If the user inputs 3 invalid codes consecutively the user will be locked out for a minute. During this lockout period only the administrator of the system will be able to unlock the system. The outcome was a successful security system which successfully implemented all of the necessary functionalities as well as incorporating a few new twists on the design.

Homework

1. The keypad scanning routine given in Lab 3 uses a scheme where the upper and lower nibbles of an i/o register are alternately used for input and output. Figure out a way to scan a 4x4 keypad which drives the vertical wires as 4 output bits and reads the horizontal wires as inputs, with no i/o swapping. Does either scheme have an advantage?

Another way to implement keyboard scanning is to set half of the port as an input port and the other half as an input. The output is wired up to the vertical pins of
the keypad and the input side of the port is wired up to the horizontal pins on the keypad. Then all that needs to be done is a 1 needs to be scanned through the output side of the port (the rest of the bits set to 0) and the input side of the port is then representative of any buttons being pressed on the row that the one is on. This is then done for all of the rows of the keypad. In this way we can find the state of the keypad. This scheme is the more advantageous of the schemes. It allows for faster scanning times and a much more efficient way to check for multiple simultaneous key presses. We chose to stick with our old keypad scanning code for the convenience of time and the fact that time precisions, as well as multiple key presses, were not of concern for our application in this lab.

2. Discuss how you will implement the required day:hrs:min clock. The scheme must be as accurate as the crystal.

We implemented the clock in a similar fashion as we did in Lab one. We define a variable ticker to be a measure of milliseconds, such that every 1000 ticks, we know a second has passed. This is repeated, until minutes, and hours have been passed. We have also implemented a calendar function, to determine which day it is. The date is initialized on January 1st, 2006, but the security person can change that through the function setDate( ). So once 31 days have been passed, it will go back to day 1 for February, and so on. Also we implemented a leap year function.
reminiscent of the UNIX cal command that will accurately set the days in February depending on the year.

3. How accurate is the crystal? Find a data sheet.

Through Digikey.com, we found that the crystal is accurate to within ±0.005% frequency at 25°C. The data sheet can be found at [link to data sheet].

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency Range</td>
<td>1.0 - 48 MHz</td>
</tr>
<tr>
<td>Calibration Tolerance</td>
<td>±0.005% (±50 ppm) at +25°C</td>
</tr>
<tr>
<td>Temperature Stability</td>
<td>±0.005% (±50 ppm) of +25°C reading</td>
</tr>
</tbody>
</table>

**Design and Test**

**Initial Setup:**

**Hardware**

The initial hardware set-up consisted of the MCU set up in a STK500. Port C was used for the keypad and wired up appropriately on a nearby breadboard. This keypad was the standard one used in lab and can be seen below,

(a) Each switch shorts one row to one column.
(b) Each pin should be connected to one bit of an i/o port.
(c) The i/o port pins will be used both as inputs and outputs. When they are inputs, they should have active pullup resistors.
(d) On the 9-pin models, do not connect the common lead.

For temporary debugging purposes we used port A to drive the LEDs so that we could make sure that the inputs demonstrated in hyperterm were agreeing with the inputs we
were entering into the keypad. Other than this we used a two wire connector from Port pins D.0 and D.1 to the RS232 jumper.

**Software**

The software set-up was the same as all previous labs. Code was written mainly in C in the CodeVisionAVR IDE. Code was mainly written before the lab period and the following header files were included and used for the indicated purposes:

```c
#include <mega32.h>

This was used for the MCU defines and register names. This is the standard header that is included for projects using the Mega32 MCU.

#include <delay.h>

This was used for the delay_us( ) function for the keypad scanning code.

#include <stdio.h>

This was included for the purpose of a few important io functions such as sprintf, sscanf, and printf. This is a pretty standard library include in c.

**Design**

**Design of the Hardware**

The design of the hardware was basically given to us in the lab assignment. No strange designs were incorporated into our project. All of the work we did was software based. The hardware design was simply a computer interface to the CPU and a keypad wired up to the CPU such that both could be used at the same time.

**Design of the Software**

The design of the software was pretty intuitive. First we focused on input and output working in both the keypad and the UART connection. After we had this working as non-blocking functions implementing the actual interface was rather easy. We also
implemented some higher-level security features that are described below. The functions are listed and their functionality described.

```c
void initialize(void)

    This function was the first block of our program to be written. This function initializes the UART, Port C for the keypad, and Timer0 for the use of the timing that controls the date and time. It also initializes the keypad state array to nopush for all of the keys such that no unpredictable states are seen in the keypad state array.
```

```c
interrupt [TIM0_COMP] void timer0_overflow(void)

    We used the counter to count in millisecond increments. This was used to increment the time scales and internal variables which determined the time of day and date.
```

```c
interrupt [USART_RXC] void uart_rec(void)

    This interrupt is called whenever the UART receives a character from the hyperterm. When the return key is received the interrupt flags the r_ready variable so that the program knows that the r_buffer is now readable. This is later used in checkRX() to check for an input command from the server master user.
```

```c
interrupt [USART_DRE] void uart_send(void)

    This interrupt works in conjunction with the puts_int() function. Basically it sends all of the characters of t_buffer to the UART to send to hyperterm. When it is done it kills the ISR ends the transmission.
```

```c
void gets_int(void)

    This function works in conjunction with the receive ISR. It resets the r_ready variable and the r_index variable so that the next receive is properly received and turns on the receive ISR so that it is ready to receive a new string for the next time.
```
void puts_int(void)

This function worked in conjunction with the transmit ISR. It basically sets up the ISR and transmits one character. Once this is done the ISR kicks in and does the rest of the work.

char keypadInt(void)

This function basically returns the character representation of the key press just entered. This function scans the keypad state table to find the first element that is ‘pushed’ and then returns its character representation. This has the downside that only one key can be recognized at a time (as our keypad scanning code does since we did not implement it bitwise) but this is not important since the assignment stated that only one key press need be assessed at any given moment and multiple key-presses should be dealt with in our own fashion. This scheme does have the advantage of a predictable output every time the same buttons are pressed down at the same time.

void updateKeys(void)

As in many of the previous labs a state machine had to be constructed for the debouncing of the buttons on the keypad. This function does that as well as scans the keypad as to update the keypad state array. The only real difference between this function and those used in previous labs is that we scanned through all of the 16 keys and did so always, even when a key press was not triggered. This difference showed us where a very annoying bug in lab 2 was coming from, which we did not have to deal with in this lab due to this alteration. Another difference is that the action done by a key was not called from this function anymore for increased functionality in the code. These are simply improvements to the previous design that this function was based on.

void updateDate(void)

This function simply updated the internal state of the timing variables to properly and accurately increment the date and time. This function is called ever 1000 milliseconds and starts off by incrementing the seconds and this will cause any number of other variables to change such as day, month etc. We implemented support for leap years
which is reminiscent of the UNIX cal command and has the same pitfalls due to the fact that when the catholic church began to count leap years it did so in increments of 3 years instead of 4 accidentally and had to make up for this with 5 year leap years for a short period thereafter.

At this point the functions are split into user client side functions and server side functions. The client side refers to the keypad interface and the server side refers to the master user manipulating the system from hyperterm.

**Client Side**

```c
void updateSys(char inputKey) {
    There was only one real client side function that was the one that handled input and compared it to our list of codes. We allowed a maximum number of 16 codes total and the program comes with 8 codes coded in as was designated on the lab assignment sheet. We decided that it would be very CPU intensive to compare every character of every code with every character of the inputted code since this could potentially compare 16 invalid codes redundantly as well as having the potential of regarding 0000 as a correct code when this was our base code to initialize invalid code and we also wanted to present 0000 as a potential code for not the smartest of users. To do this we implemented a codeValid array which was an array of single digits that represented whether that specific code was valid with a 0 for invalid and 1 for valid. These not only helped us with the implementation of code checking but also with manipulating the code array to add, delete or view the codes.

    The code then found that the code was correct or not. In the case that it was not correct an attempt variable was incremented. When this variable reached 3 the client side functions were not allowed to run thusly locking the client out. In either case the lockout time (which is the time for which we keep any client side functionality from occurring) was set to a certain amount of time (3 seconds for non-lockout and a minute for lockout) and LED7 turned on. If the code was correct then the attempt variable was reset to zero, the lockout time set to 3 seconds (this allows us to ensure that the system doesn’t lock up the system in the middle of an unlock) and LED0 set.
```
The LEDs were reset in the main function every main loop unless the lockout time was greater than 0. This sort of implementation allowed us to maintain that all of the functionality of the client side was correct and not interrupted by any sort of client input. The server side is still running however and could easily lockout a system at the moment right after it had been unlocked. This allows the client side to be non-blocking.

**Server Side**

```c
void checkRX(void)
```

This function was the entry point for the server-side coding. This function is extremely simple. It checks for an input waiting in the UART and if one is there it uses the ISR to download this input into the transmit buffer. This buffer is then compared to the many server side functions that are possible. We had two types of server side functions. One function that took no parameters which after was called the ‘current command’ variable was reset to no command and the specific function run (such as the help function or print list).

Other functions required parameters. These inputs would set the current command variable to themselves that way the next time checkTX( ) was called it knew that it was waiting for a parameter input not a new command. When these parameters were inputted then the function would call the appropriate function with these parameters as inputs. Examples of these functions are setDate(m/d/y) and addCode(code to be added). Correct debugging of this function early in the process allowed us to implement these functions very quickly as well as potentially adding as many functions as we wanted. A few very useful defines (in conjunction with a server side function help( ) that simply listed all of the functions ) helped us to determine which function calls were which:

```c
#define nocommand 0
#define Unlock 1
#define lockout 2
#define setdate 3
#define addcode 4
#define delcode 5
```
It is important to note that for the lockout functionality no function needed to be implemented. It was as simple as setting the attempt variable to any value equal or greater than 3. This was done in a quick line when the lockout command was registered as setting attempt to the value of 4. The unlock function then easily overwrites this. LED7 was then programmed to turn on whenever attempt was greater than or equal to 3 that way we could ensure that even in the case of a permanent lockout LED7 was turned on.


unsigned char unlock(unsigned long a, unsigned long b)

This function was potentially very simple but we decided to implement a more interesting security algorithm to provide better security. It is possible for a hacker to break into a program’s assembly code and find out what a hardwired code is to force the unlock of a system. In our program it’s actually defined at the top of the program:

#define key 50424113

However this is not the code to unlock the system. One must present two numbers to the system. These two numbers when multiplied need to be equal to the preprogrammed code. This seems ‘not that bad’ but the two numbers are actually rather large prime numbers and to factorize a number such as the one above into the correct prime numbers, even when one knows all of the prime numbers between 1 and 50424113 would take a long time. This is a very simple and robust cryptology scheme to allow very good server side security.

void printList(void)

This function prints the list of valid codes to hyperterm. Notice that only valid codes are printed and all the others are omitted. The list output format is as follows:
code #) code
void addCode(char code[4])

This function finds the first code slot that is invalid (meaning not taken) and
inserts the master user’s new code[4] into the list of codes. It then flips the validCode
array element that refers to this code slot. This adds a new valid code to the last of
possible codes.

void deleteCode(char num)

This function deletes the code at the user defined slot num. This number can be
found from the printList() function for ease of manipulating the code list.

void setDate(unsigned char m, unsigned char d, unsigned int y)

This function takes three parameters and passes them to the program’s internal state
variables that define the current date.

void helplist(void)

This function outputs a list of available functions to be used in hyperterm for
the server-side of the system.

void dispDate(void)

This function, which is not listed in helplist(), is simply there for the purpose of
debugging the setDate() function. To use it an 8 is inputted at hyperterm.

void main(void)

This is the entry point into our program. This essentially takes all of the
functions described above and strings them together to produce a working program. First
it calls initialize() to initialize all of the internal variables and globals. It then enters the
main program loop. Within the program loop it checks to see when uptime is equal to 0.
If it is then it resets uptime and runs the program code. This was done so that the
program doesn’t poll the UART all the time and essentially slows down the program a
little so that we can get more reliable results especially when checking the key pad states.
Next it checks the state of the keys using updateKeys( ) and stores the ascii value of the input key if one exists. Then main sets up the lastlastkey and lastkey variables accordingly such that a key held down does not mean that key pad press over and over again. This was a scheme implemented in lab 2 and is more robust in this application of it since it applies to more functions rather than just one where it was locally used.

Main then checks the ticks value (which is updated via the timer ISR). When this reaches a value equivalent of a second it resets the ticks and calls the updateDate( ) function. Next main checks the for input from the master user with checkRX( ). At this point in the main program loop the program has already taken care of server side operations and moves on with client side. This is ideal since we want server-side operations to take precedence in the case that they occur in the same program loop. This means that both client and server side operations are valid in one program loop but in the case that a code is deleted on the server side as it is entered on the client side it will mean that the code will be invalid. This is preferable in our implementation.

According to the internal state variables such as attempt, lastkey, lastlastkey, and lockoutTicks (which are all set by the different functions) the program will call the updateSys( ) function. The rest of the functionality in the main program loop has to do with changing the states of the LEDs according to the value of attempt and lockoutTicks.

Testing, Hurdles and Solutions

Testing our design proved very successful and after we had gotten all of the input and output working not too many bugs were present. One of the notable bugs that we had had to do with the keypad state table not being updated due to some redundant code and once this was taken care of no real coding problems were noticed other than getting the LED to light up at the right times (this was taken care of in the main program loop and depended on the value of attempt. Using an internal program variable proved itself as a very good design since the LEDs were being updated in real time according to a variable that could be manipulated rather than manipulating the LEDs directly).

The biggest problem that we ran into was the fact that the t_buffer and r_buffer were actually overwriting our codeValid array. This was a memory error and was fixed
by putting the TX and RX variables at the end of global variable declarations. Due to the strange and esoteric nature of this bug this was an easy fix. In future applications memory management needs to be used.

**Questions**

1) We used the same scanning code scheme as used in Lab 2 with a few improvements. This saved us some time and allowed us to focus on more important aspects of the design such as the interface and UART.

2) To convert the inputted code to an ASCII value we used a switch statement that returned a certain character for each keypad position which was then in the pushed state. This was all contained in the keypadInt( ) function described above. Some more eloquent methods exist such as adding 48 to the value of the keypad state table index that was in a pushed state in the case that it was between 1 and 9. This simply encodes it into direct ascii code. Then we can use similar conditionals for 0 and the letters since they are all in sequential order.

→ For 0 return ‘0’. For keys 11 through 16 return (65 + (indexValue – 11)). This would return A for 11 and C for 13 for example.

**Conclusions and observations:**

This lab was successful in that all requirements of the security system were met. We encountered some very esoteric memory allocation errors that were corrected by rearranging the code to prevent any variables being rewritten by the RX/TX buffers. We implemented a more secure way for the master user to unlock the system using some prime number cryptology. We feel that we could have made the menu easier to navigate by using submenus or embellishing somewhat on the format of commands and their parameters. This would be very useful in the deleteCode( ) function since the master user can accidentally delete the wrong code if they don’t use the printlist command. This lab could be expanded in several ways. For example, we could generate a tone each time a digit is pressed (same tone for each digit) so that the user knows whether the number was
fully pressed or not. Furthermore we could have implemented delete and enter button. This would be most useful if an LCD display was used to display the code being entered and every time a digit is pressed a ‘*’ would be displayed or the actual number as defined by the master user in an internal state variable in the program. This way the user can have more reliability in their input of codes into the system that are of correct length and in the case that the user is not sure of the code that they have entered they can change their minds before pressing enter, indicating that this is the final code to be used.