|  |
| --- |
| **Programmable IR-Remote Controlled Locking Actuator**  Freddy Aguinaga  12/05/2019 |

**Theory of Operation**

The programmable IR-Remote Controlled Lock Actuator accepts two major inputs to trigger the locking and unlocking features of the device: the IR remote and the UART interface.

**IR Remote**

**What the IR remote signal consists of**

The IR remote sends an infrared signal to the IR receiver which generates a signal that can be divided into 3 parts: the preamble, identifying bits, and lastly the data bits.

The preamble has varying periods between transitions from a logic 1 to a logic 0 and vice versa. It relies on this variation to differentiate from other IR controllers. In between the preamble and the identifying bits there are toggling bits that determine whether a new button was pressed if the toggling bits have changed or if the button is being held down where the toggling bits would remain the same. Next, the identifying bits are a series of unchanging bits that also help confirming much like the preamble so I ultimately grouped them up. Finally, the data bits at the end of the signal change according to the button pressed which I use to make a button code.

**How I Process the IR Signal**

A GPIO pin named GPI was used to take the input signal from the IR receiver to be processed by interrupting in the middle of each logic 1 or logic 0 states to confirm it is from the specific remote I am using. I accomplished this by first having the GPI pin generate an interrupt whenever it first notices a negative edge occur. After the first edge interrupt, the function turns on a timer which will take values of the time difference of each transition divided by 2 to interrupt in the middle of each logic 1 and logic 0 state in the preamble and identifying bits. Once I have reached the data bit portion of the signal, I use a constant value that interrupted in the middle of each bit instead of the whole on/off state. By declaring a 16 bit variable, I was able to store the bits by storing one bit at a time and shifting it left in the variable. This was done because the data bits in the signal are read from the most significant bit to the least significant bit. After all 16 bits have been stored, I can then determine what button is pressed by the number produced.

**Action After Generating the Button Code**

When the signal is completely processed and a code is generated, it is then compared with the EEPROM’s 32 blocks (each with 16 words) which may or may not contain password sequences of at most 15 buttons long. Each of these blocks will contain the length of the password at the beginning of each block at word 0. Starting from word 1 for all blocks, I check if the key pressed is present in at least one of the passwords. If there is a match, I store the button pressed in a sequence buffer and iterate to the next word in the block. When I find matches in the later words, I will confirm the previous values of our buffer with the previous values in the block to make sure they both match to confirm there is a possible password match. After each match, I check our index value with the password length at word 0 to see if I have completed the password sequence. This way I always find the shortest password without relying on the ordering of passwords in the EEPROM. After a password sequence has been entered, the device will lock or unlock if it is a block between 0-30 or panic if it is a password in block 31.

**UART Interface**

The UART interface was created to configure the device’s capability in lock or unlocking by sending commands to the device through a computer. It could also configure the device’s own time module, LEDs, and button functionality.

**Accepting Commands From UART**

The UART being used by the board accepts the characters sent by the computer by placing them in the receiving FIFO which are later taken out to be stored in a character array. The array is then divided up to four times at points where there are whitespaces present. The location of the first letter of each word is stored in a char pointer array to be processed to execute commands. The first word determines what command is going to be executed and the following words (if any) act as arguments of that command. This is also the only way to set a new password by setting the setPass variable and accepting all the following buttons pressed by the IR remote and storing them in the user-defined block.

Valid Commands

lock – Lock the device

unlock – Unlock the device

status – Provide status of locked, unlocked, or jammed

gettime – Prints current time of the device

clear – Clears all EEPROM blocks of passwords

password {P} – Sets password of max length of 15 keys for 10 seconds

erase {P} – Sets panic sequence of max length of 15 keys for 10 seconds

local {ON|OFF} – Enable/disable local button functionality to lock or unlock

beeper {ON|OFF} – Enable/disable sound for every button press

led {ON|OFF} – Enable/disable LED indicators of status for the device

time {H} {M} – Set the system time for the device

autolock {H,M} | disable – Set/disable the time the device will auto lock

**Using Hibernation Module for Auto Locking**

In order to implement the feature of auto locking. I needed to turn on the hibernating module only to use RTC feature of the module. I first turned it on by setting the clocking and enable bit in the control register. After being turned on, I then load the default value of 0 seconds. The RTCM0 or seconds matching register, is set to a large number that the RTCC register will not reach since the register is reset to 0 when it reaches 24 hours in seconds. The WRC interrupt is set to actively keep track of the RTC time when the module is not busy. After configuration, I can set the times after the receiving the command from the UART. After receiving the arguments H (hours) and M (Minutes), I convert them to seconds (using the formulas shown in the next page) and input the sum into the RTCM0 register and enable interrupts for matches.

Hours = Input\_hours\_index \* 3600

Minutes = Input\_minutes\_index \* 60

\* placeholder indicates if the number is in the ones or tens place.

**Using the Beeper, LEDs, and Local features**

The beeper, LEDs, and local features are set to individual GPIO pins that are bit-banded and can simply be turned on or off. The beepEnable variable is made specifically for the beeper because the function I made that will toggle the beeper pin to make sound can easily be bypassed checking the variable if it is set to true or false. The local pin allows for a physical button on the board to activate the locking/unlocking mechanism of the board. This can be disabled by disabling interrupts made by the button. LEDs are purely controlled by the pre-assigned pins in port F. Depending on the status of the board, it will indicate whether it is locked (red) and unlocked (green).

**Conclusion**

Through this project I was able to learn different concepts of timing and interrupts and the difficulty associated with them. How tools such as the expression and variable viewing tools and breakpoints help determine where problems are occurring. Also, by having a solid plan made before, programs are more easily implemented by having expectations of what types of input and outputs are needed.

**#include** <stdint.h>

**#include** <stdio.h>

**#include** <stdbool.h>

**#include** <assert.h>

**#include** "tm4c123gh6pm.h"

//bit band stuff

**#define** GPI (\*((**volatile** uint32\_t \*)(0x42000000 + (0x400053FC-0x40000000)\*32 + 0\*4)))

**#define** GPO (\*((**volatile** uint32\_t \*)(0x42000000 + (0x400053FC-0x40000000)\*32 + 5\*4)))

**#define** SENSOR (\*((**volatile** uint32\_t \*)(0x42000000 + (0x400053FC-0x40000000)\*32 + 3\*4)))

**#define** SPEAKER (\*((**volatile** uint32\_t \*)(0x42000000 + (0x400063FC-0x40000000)\*32 + 4\*4)))

**#define** RED\_LED (\*((**volatile** uint32\_t \*)(0x42000000 + (0x400253FC-0x40000000)\*32 + 1\*4)))

**#define** GREEN\_LED (\*((**volatile** uint32\_t \*)(0x42000000 + (0x400253FC-0x40000000)\*32 + 3\*4)))

**#define** PUSH\_BUTTON (\*((**volatile** uint32\_t \*)(0x42000000 + (0x400253FC-0x40000000)\*32 + 4\*4)))

**#define** PANIC\_BUTTON (\*((**volatile** uint32\_t \*)(0x42000000 + (0x400253FC-0x40000000)\*32 + 0\*4)))

//Motor pins bit band

**#define** RED (\*((**volatile** uint32\_t \*)(0x42000000 + (0x400063FC-0x40000000)\*32 + 6\*4)))

**#define** WHITE (\*((**volatile** uint32\_t \*)(0x42000000 + (0x400063FC-0x40000000)\*32 + 5\*4)))

**#define** YELLOW (\*((**volatile** uint32\_t \*)(0x42000000 + (0x400053FC-0x40000000)\*32 + 6\*4)))

**#define** BLUE (\*((**volatile** uint32\_t \*)(0x42000000 + (0x400063FC-0x40000000)\*32 + 7\*4)))

//Port F Button and LEDs

**#define** PANIC\_BUTTON\_MASK 1

**#define** BUTTON\_MASK 16

**#define** RED\_LED\_MASK 2

**#define** GREEN\_LED\_MASK 8

//MOTOR pins

**#define** RED\_MASK 64 //PC6

**#define** WHITE\_MASK 32 //PC5

**#define** YELLOW\_MASK 64 //PB6

**#define** BLUE\_MASK 128 //PC7

//Speaker pin

**#define** SPEAKER\_MASK 16 // pin PC4 for speaker output

//Port B pins for IR sensor & light sensor

**#define** SENSOR\_MASK 8 // pin PB3 mask for status mechanism

**#define** GPI\_MASK 1 //pin PB0 input from IR receiver

**#define** GPO\_MASK 32 // pin PB5 output for IR receiver

**#define** delay4Cycles() **\_\_asm**(" NOP\n NOP\n NOP\n NOP")

//NOTE: Timing values may differ from original excel sheet

// 41th and 42th element captures are ignored

// 44th element last of preamble

**const** uint32\_t preamble\_LD[] =

{

54000, 17400, 9000, 12160, 6220, //5

7820, 10080, 16780, 10080, 16460, //10

28000, 16520, 9520, 8360, 10000, //15

7860, 10080, 7820, 10080, 7420, //20

10080, 7820, 10100, 7800, 10080, //25

7840, 10560, 7340, 10080, 7840, //30

10060, 7840, 18040, 7920, 10080, //35

7820, 10080, 7820, 10500, 7000, //40

7820, 12000, 8000, 7200, 7350, //45

21200, 8140, 8140, 16480, 19320, //50

10640, 6900, 10400, 7100, //54

};

**const** uint8\_t preamble\_size = 54; // size of preamble

uint8\_t local = 1; //Local button default cannot unlock/lock

uint8\_t panic = 0;

// Beeper variables

uint8\_t beepEnable = 0; // beeper enable option

**int** status = 0;

// IR variables

uint8\_t w\_index = 1; //current location in buffer password

uint8\_t password\_length = 16; //

uint8\_t setPass = 0; //not currently setting password

uint8\_t phase = 0; //phase counter

uint16\_t code = 0; //controller button code value

uint32\_t prev = 0; //previous load value

uint32\_t curr = 0; //current load value

**void** **initHw**()

{

// 40MHz Clock, PLL enabled

SYSCTL\_RCC\_R = SYSCTL\_RCC\_XTAL\_16MHZ | SYSCTL\_RCC\_OSCSRC\_MAIN | SYSCTL\_RCC\_USESYSDIV | (4 << SYSCTL\_RCC\_SYSDIV\_S);

//Clocks for GPIO ports and Timer

SYSCTL\_RCGC2\_R |= SYSCTL\_RCGC2\_GPIOF | SYSCTL\_RCGC2\_GPIOC | SYSCTL\_RCGC2\_GPIOB | SYSCTL\_RCGC2\_GPIOA;

SYSCTL\_RCGCTIMER\_R |= SYSCTL\_RCGCTIMER\_R1 | SYSCTL\_RCGCTIMER\_R2;

SYSCTL\_RCGCEEPROM\_R = 1;

SYSCTL\_RCGCHIB\_R |= 1;

// Set Ports to APB

SYSCTL\_GPIOHBCTL\_R = 0;

//enable board LEDS and button

GPIO\_PORTF\_LOCK\_R = GPIO\_LOCK\_KEY;

GPIO\_PORTF\_CR\_R |= PANIC\_BUTTON\_MASK;

GPIO\_PORTF\_DIR\_R |= GREEN\_LED\_MASK | RED\_LED\_MASK; // bits 1 and 3 are outputs, other pins are inputs

GPIO\_PORTF\_DR2R\_R |= GREEN\_LED\_MASK | RED\_LED\_MASK; // set drive strength to 2mA (not needed since default configuration -- for clarity)

GPIO\_PORTF\_DEN\_R |= PANIC\_BUTTON\_MASK | BUTTON\_MASK | GREEN\_LED\_MASK | RED\_LED\_MASK; // enable LEDs and pushbuttons

GPIO\_PORTF\_PUR\_R |= PANIC\_BUTTON\_MASK | BUTTON\_MASK; // enable internal pull-up for push button

GPIO\_PORTF\_IS\_R &= ~BUTTON\_MASK;

GPIO\_PORTF\_IBE\_R &= ~BUTTON\_MASK;

GPIO\_PORTF\_IEV\_R &= ~BUTTON\_MASK;

GPIO\_PORTF\_ICR\_R |= BUTTON\_MASK;

GPIO\_PORTF\_IM\_R |= BUTTON\_MASK;

NVIC\_EN0\_R |= (1 << 30); // enable port F interrupts

**if**(SENSOR)

{

RED\_LED = 0;

GREEN\_LED = 1;

status = 0;

}

**else**

{

RED\_LED = 1;

GREEN\_LED = 0;

status = 1;

}

//enable speaker output pin PC4

GPIO\_PORTC\_DIR\_R |= SPEAKER\_MASK | RED\_MASK | BLUE\_MASK | WHITE\_MASK; // set speaker pin as output

GPIO\_PORTC\_DEN\_R |= SPEAKER\_MASK | RED\_MASK | BLUE\_MASK | WHITE\_MASK; // enable digital

GPIO\_PORTC\_DR2R\_R |= SPEAKER\_MASK; // limit current to 2mA

//enable GPI interrupt (Falling edge) and GPO pins

GPIO\_PORTB\_DIR\_R |= GPO\_MASK | YELLOW\_MASK;

GPIO\_PORTB\_DIR\_R &= ~SENSOR\_MASK;

GPIO\_PORTB\_DEN\_R |= GPI\_MASK | GPO\_MASK | SENSOR\_MASK | YELLOW\_MASK;

GPIO\_PORTB\_PUR\_R |= SENSOR\_MASK;

GPIO\_PORTB\_IS\_R &= ~GPI\_MASK;

GPIO\_PORTB\_IBE\_R &= ~GPI\_MASK;

GPIO\_PORTB\_IEV\_R &= ~GPI\_MASK;

GPIO\_PORTB\_ICR\_R |= GPI\_MASK;

GPIO\_PORTB\_IM\_R |= GPI\_MASK;

// Configure pins for Tx and Rx

GPIO\_PORTA\_DIR\_R |= 2; // enable output on UART0 TX pin: default, added for clarity

GPIO\_PORTA\_DEN\_R |= 3; // enable digital on UART0 pins: default, added for clarity

GPIO\_PORTA\_AFSEL\_R |= 3; // use peripheral to drive PA0, PA1: default, added for clarity

GPIO\_PORTA\_PCTL\_R &= 0xFFFFFF00; // set fields for PA0 and PA1 to zero

GPIO\_PORTA\_PCTL\_R |= GPIO\_PCTL\_PA1\_U0TX | GPIO\_PCTL\_PA0\_U0RX;

// select UART0 to drive pins PA0 and PA1: default, added for clarity

// Configure UART0 to 115200 baud, 8N1 format

SYSCTL\_RCGCUART\_R |= SYSCTL\_RCGCUART\_R0; // turn-on UART0, leave other UARTs in same status

delay4Cycles(); // wait 4 clock cycles

UART0\_CTL\_R = 0; // turn-off UART0 to allow safe programming

UART0\_CC\_R = UART\_CC\_CS\_SYSCLK; // use system clock (40 MHz)

UART0\_IBRD\_R = 21; // r = 40 MHz / (Nx115.2kHz), set floor(r)=21, where N=16

UART0\_FBRD\_R = 45; // round(fract(r)\*64)=45

UART0\_LCRH\_R = UART\_LCRH\_WLEN\_8 | UART\_LCRH\_FEN; // configure for 8N1 w/ 16-level FIFO

UART0\_CTL\_R = UART\_CTL\_TXE | UART\_CTL\_RXE | UART\_CTL\_UARTEN; // enable TX, RX, and module

//Configure periodic timer and leave off

TIMER1\_CTL\_R &= ~TIMER\_CTL\_TAEN; //Turn off timer

TIMER1\_CFG\_R |= TIMER\_CFG\_32\_BIT\_TIMER; //Timer with A&B to make 32 bit timer

TIMER1\_TAMR\_R |= TIMER\_TAMR\_TAMR\_1\_SHOT; //one shot timer

TIMER1\_TAILR\_R = 40000000; // default value

TIMER1\_IMR\_R |= TIMER\_IMR\_TATOIM; // turn on interrupts for timer1

//Enable RTC

HIB\_CTL\_R |= HIB\_CTL\_CLK32EN; // Turn on RTC

**while**(!(HIB\_CTL\_R & HIB\_CTL\_WRC));

HIB\_IC\_R |= 0x11; // Clear interrupt

**while**(!(HIB\_CTL\_R & HIB\_CTL\_WRC));

HIB\_CTL\_R |= HIB\_CTL\_RTCEN;

**while**(!(HIB\_CTL\_R & HIB\_CTL\_WRC)); // wait until it is not busy

HIB\_RTCLD\_R = 0; // Load seconds in

**while**(!(HIB\_CTL\_R & HIB\_CTL\_WRC));

HIB\_RTCM0\_R = 4094967296; // Just a really high number that it shouldn't reach

**while**(!(HIB\_CTL\_R & HIB\_CTL\_WRC));

HIB\_RTCSS\_R = 0; // set subseconds match

NVIC\_EN1\_R |= (0x1 << 11);

**while**(!(HIB\_CTL\_R & HIB\_CTL\_WRC));

HIB\_IM\_R |= HIB\_IM\_WC | HIB\_IM\_RTCALT0; //turn on interrupts

**while**(EEPROM\_EEDONE\_R & EEPROM\_EEDONE\_WORKING); //Wait until EEPROM is not powering up/working

**if**(EEPROM\_EESUPP\_R & (EEPROM\_EESUPP\_PRETRY || EEPROM\_EESUPP\_ERETRY))

{

assert(0);

}

SYSCTL\_SREEPROM\_R = 1; //reset EEPROM

SYSCTL\_SREEPROM\_R = 0; //clear reset

//

}

**void** **waitMicrosecond**(uint32\_t us)

{

**\_\_asm**("WMS\_LOOP0: MOV R1, #6"); // 1

**\_\_asm**("WMS\_LOOP1: SUB R1, #1"); // 6

**\_\_asm**(" CBZ R1, WMS\_DONE1"); // 5+1\*3

**\_\_asm**(" NOP"); // 5

**\_\_asm**(" NOP"); // 5

**\_\_asm**(" B WMS\_LOOP1"); // 5\*2 (speculative, so P=1)

**\_\_asm**("WMS\_DONE1: SUB R0, #1"); // 1

**\_\_asm**(" CBZ R0, WMS\_DONE0"); // 1

**\_\_asm**(" NOP"); // 1

**\_\_asm**(" B WMS\_LOOP0"); // 1\*2 (speculative, so P=1)

**\_\_asm**("WMS\_DONE0:"); // ---

// 40 clocks/us + error

}

**int** **strcmp**(**char**\* str1, **char**\* str2)

{

**int** sum = 0;

**while**(\*str1 != '\0' && \*str2 != '\0')

{

sum = \*str1 - \*str2;

**if**(sum != 0)

**break**;

str2++;

str1++;

}

**return** \*str1-\*str2;

}

**int** **strlen**(**char**\*str1)

{

**int** length = 0;

**while**(\*str1 != '\0')

{

str1++;

length++;

}

**return** length;

}

// Blocking function that writes a serial character when the UART buffer is not full

**void** **putcUart0**(**char** c)

{

**while** (UART0\_FR\_R & UART\_FR\_TXFF); // wait if uart0 tx fifo full

UART0\_DR\_R = c; // write character to fifo

}

// Blocking function that returns with serial data once the buffer is not empty

**char** **getcUart0**()

{

**while** (UART0\_FR\_R & UART\_FR\_RXFE); // wait if uart0 rx fifo empty

**return** UART0\_DR\_R & 0xFF; // get character from fifo

}

// Blocking function that writes a string when the UART buffer is not full

**void** **putsUart0**(**char**\* str)

{

uint8\_t i;

**for** (i = 0; i < strlen(str); i++)

putcUart0(str[i]);

}

**void** **getsUart0**(**char**\* str, uint8\_t maxChars)

{

**char** letter;

uint8\_t count=0;

**while**(count < maxChars)

{

letter = getcUart0();

// return if enter is pressed

**if**(letter == 13)

{

**break**;

}

// Checks if letter is a backspace ascii number

**else** **if**(letter == 8 || letter == 127)

{

**if**(count > 0)

{

count--;

putsUart0("\b \b");

}

}

//if character is printable, add to string

**else** **if**(letter >= ' ')

{

//if character is uppercase letter, convert to lower case

**if**(letter >= 'A' && letter <= 'Z')

{

letter += 32;

}

putcUart0(letter);

str[count] = letter;

count++;

}

}

// if the count == maxChars, end with null terminator and return

str[count] = '\0';

}

**void** **print**(**char**\* str)

{

putsUart0(str);

putcUart0(10);

putcUart0(13);

}

uint8\_t **parseStr**(**char**\* str, **char**\*\* words, uint8\_t maxChars, uint8\_t maxWords)

{

**if**(str == NULL || words == NULL)

{

print("ERROR: parseStr invalid argument");

**return** 0;

}

**int** i;

**int** w\_index = 1;

words[0] = &str[0]; // first word is the first argument

**for**(i = 0; i < maxChars && str[i] != '\0'; i++)

{

**if**(w\_index >= maxWords)

{

**break**;

}

**if**(str[i] == ' ')

{

str[i] = '\0';

**if**(str[i+1] != 0 && str[i+1] != '\0' && str[i+1] != ' ')

{

words[w\_index] = &str[i+1];

w\_index++;

}

}

}

**return** w\_index;

}

**void** **beep**()

{

**if**(beepEnable)

{

**int** milliseconds = 50;

**while**(milliseconds > 0)

{

SPEAKER ^= 1;

waitMicrosecond(1000); // wait 1ms

milliseconds--;

}

SPEAKER = 0;

}

}

**int** **lock**()

{

//if it is already locked, then remain locked

**if**(!SENSOR)

{

status = 1;

**return** status;

}

//if no leads are powered, Red is default

**if**(!RED && !WHITE && !YELLOW && !BLUE)

{

RED = 1;

}

**int** i;

**for**(i = 0; i < 120; i++)

{

waitMicrosecond(6000); // step per microsecond

**if**(RED)

{

RED = 0;

YELLOW = 1;

}

**else** **if**(YELLOW)

{

YELLOW = 0;

WHITE = 1;

}

**else** **if**(WHITE)

{

WHITE = 0;

BLUE = 1;

}

**else** **if**(BLUE)

{

BLUE = 0;

RED = 1;

}

**else**

print("Motor error: No motor leads are powered.");

}

//check if motor is still unlocked after trying to lock

**if**(SENSOR)

{

GREEN\_LED = 1;

RED\_LED = 0;

status = -1; // jammed

}

**else**

{

GREEN\_LED = 0;

RED\_LED = 1;

status = 1; // locked

}

**return** status;

}

**int** **unlock**()

{

//check if it is unlocked

**if**(SENSOR)

{

status = 0;

**return** status;

}

**int** i;

**if**(!RED && !WHITE && !YELLOW && !BLUE)

{

RED = 1;

}

**for**(i = 0; i < 120; i++)

{

waitMicrosecond(6000); // step per microsecond

**if**(RED)

{

RED = 0;

BLUE = 1;

}

**else** **if**(YELLOW)

{

YELLOW = 0;

RED = 1;

}

**else** **if**(WHITE)

{

WHITE = 0;

YELLOW = 1;

}

**else** **if**(BLUE)

{

BLUE = 0;

WHITE = 1;

}

**else**

print("Motor error: No motor leads are powered, Resetting to 'RED'");

}

//if motor is still locked

**if**(!SENSOR)

{

GREEN\_LED = 0;

RED\_LED = 1;

status = -1; // jammed

}

**else**

{

GREEN\_LED = 1;

RED\_LED = 0;

status = 0; // unlocked

}

**return** status;

}

// temporarily enable beeper to create 1kHz signal for 1ms

**void** **scream**()

{

panic = 1;

beepEnable = 1;

**while**(PANIC\_BUTTON)

{

beep();

}

beepEnable = 0;

panic = 0;

}

**void** **edgeMode**()

{

GPIO\_PORTB\_IM\_R |= GPI\_MASK; // turn on edge triggering

TIMER1\_CTL\_R &= ~TIMER\_CTL\_TAEN; //turn off timer1A

NVIC\_EN0\_R &= ~(1 << 21); // turn off interrupts for timer1A

TIMER1\_IMR\_R &= ~TIMER\_IMR\_TATOIM; //turn off timeout interrupts in timer1A

NVIC\_EN0\_R |= 1 << 1; //turn on interrupts in port B

GPO = 0; // GPO Output starts low

phase = 0; // reset phase counter

curr = 0; // reset current load value

prev = 0; // reset previous load value

}

**void** **timerMode**()

{

TIMER1\_IMR\_R |= TIMER\_IMR\_TATOIM; // turn on countdown interrupt

NVIC\_EN0\_R |= (1 << 21); // turn on interrupt for timer1A

NVIC\_EN0\_R &= ~(1 << 1); //turn off edge triggering in portB

GPIO\_PORTB\_IM\_R &= ~GPI\_MASK; // turn off edge

TIMER1\_ICR\_R |= 1; //clear timeout interrupt in timer1

TIMER1\_CTL\_R |= TIMER\_CTL\_TAEN; // turn on timer1A

}

**void** **edgeISR**()

{

**if**(!panic)

{

curr = preamble\_LD[0];

TIMER1\_TAILR\_R = curr; // value to check first bit of data stream of 2.7ms

timerMode(); //enable timer mode to toggle in between bits

}

GPIO\_PORTB\_ICR\_R |= GPI\_MASK; // clearing interrupt at GPI pin

}

// Beeper interrupt function to toggle output at 1kHz

**void** **beepISR**()

{

SPEAKER ^= 1; //toggle beeper output to create sound

TIMER2\_ICR\_R |= 1; //clear timeout interrupt in timer2

}

**void** **timerISR**()

{

GPO ^= 1; //Toggle the bit every time.

phase++;

//verify that the signal is from our special remote

**if**(phase < preamble\_size)

{

code = 0;

TIMER1\_CTL\_R &= ~TIMER\_CTL\_TAEN; //turn off timer1

**if**(GPI == GPO && (phase < 41 || phase > 42)) // return if signal does not match ID signal components.

{

edgeMode(); //reset to edge triggering mode

TIMER1\_ICR\_R |= 1; //clear flag to leave interrupt

**return**; //exit interrupt

}

prev = curr; //set the previous half time

curr = preamble\_LD[phase]; //set current half time

TIMER1\_TAILR\_R = prev + curr; //add the previous and current half time

TIMER1\_CTL\_R |= TIMER\_CTL\_TAEN; //turn on timer1

}

//When signal has been verified and in the data bits section

**else** **if**(phase < 70)

{

code += GPI; // add the current value of signal (H/L) to create code

**if**(phase != 69) //dont shift if it is the last bit

code = code << 1; // shift to left since we are getting the data from MSB to LSB

TIMER1\_CTL\_R &= ~TIMER\_CTL\_TAEN; //turn off timer1

TIMER1\_TAILR\_R = 17600; //Set constant T for reading data bits

TIMER1\_CTL\_R |= TIMER\_CTL\_TAEN; //turn on timer1

}

//After data bits, we add

**else**

{

uint16\_t up = 43350;

uint16\_t down = 43349;

uint16\_t left = 42666;

uint16\_t right = 42665;

uint16\_t center = 42662;

uint16\_t play = 38486;

uint16\_t back = 42661;

//check if code is a valid button code.

**if**(code != up && code != down && code != left && code != right && code != center && code != play && code != back)

{

putsUart0("\n\rERROR: Invalid button code");

TIMER1\_ICR\_R |= 1;

**return**;

}

beep();

//Delay for my siblings cause they can't press buttons correctly

//waitMicrosecond(10000);

//Save buttons that have matching code in current index and verified to be same sequence so far

**static** **int** buffer[15];

//Saves button pressed in current word location in a EEPROM block

**if**(setPass)

{

//max password length is 16(0-15). All other inputs are ignored

**if**(w\_index < 16)

{

EEPROM\_EEOFFSET\_R = w\_index;

EEPROM\_EERDWR\_R = code;

w\_index++;

password\_length++;

}

}

/\*Process:

\* w\_index start from word 1, and checks each of the 32 blocks available

\* for a matching code in the current word index. Then the matching block

\* needs to be checked if the previous input buttons in the buffer match

\* the block's previous code values. From there it compares the w\_index or

\* current word index to the length of the matching block's password to see

\* if it is the smallest password with a matching sequence. Match\_block provides

\* unique passwords to be verified more quickly by saving the previous block that

\* had a matching word.

\*/

**else** **if**(w\_index < 16)

{

//Every time a button is pressed, we start searching for code match from block 0.

**static** **int** match\_block = 0;

//possible variable determines whether there is a chance of a valid

**int** possible = 0;

//i is used to store block number since EEPROM was proving to be unreliable.

**int** i = 0;

EEPROM\_EEBLOCK\_R = 0;

**while**(EEPROM\_EEDONE\_R & EEPROM\_EEDONE\_WORKING);

//Cycle through all 32 blocks including block 31(panic block)

**while**(i <= 31)

{

//Move to current word index of block to check following input code against password code.

EEPROM\_EEOFFSET\_R = w\_index;

**while**(EEPROM\_EEDONE\_R & EEPROM\_EEDONE\_WORKING);

**if**(code == EEPROM\_EERDWR\_R)

{

**if**(match\_block != EEPROM\_EEBLOCK\_R) //if code matches current column, but isn't the same block, check previous part of the sequence for matching values

{

//mismatch shows if a block does not match entirely with the current sequence (buffer)

**int** mismatch = 0;

**int** index=1;

**while**(index < w\_index) //check previous values

{

//temp used to hold word value to prevent spontaneous changes.

EEPROM\_EEOFFSET\_R = index;

**int** temp = EEPROM\_EERDWR\_R;

**if**(buffer[index-1] == temp)

{

mismatch = 0;

}

**else**

{

mismatch = 1;

**break**;

}

index++;

}

//if a block does not match entirely, do not verify as a possible match, and continue with search.

**if**(mismatch == 1)

{

**goto** MISMATCHED;

}

}

//After verifying the block matched the current sequence, save button in code sequence buffer.

buffer[w\_index-1] = code;

//Set possibility of matching password to 1/true

possible = 1;

//Check the overall password length in current block to see if we have done the complete sequence

EEPROM\_EEOFFSET\_R = 0;

**if**(w\_index >= EEPROM\_EERDWR\_R)

{

//if any blocks 0-30 are matched completely, lock/unlock

**if**(EEPROM\_EEBLOCK\_R < 31)

{

**if**(!SENSOR)

{

unlock();

}

**else** **if**(SENSOR)

{

lock();

}

}

//block 31 is reserved for panic code sequence

**else**

{

scream();

}

//Reset to first word of every block

//first value in buffer is 0 to reset because no button pressed will be zero.

w\_index = 1;

possible = 0;

buffer[0] = 0;

}

}

MISMATCHED:

//iterate through all 32 blocks

i++;

**while**(EEPROM\_EEDONE\_R & EEPROM\_EEDONE\_WORKING);

EEPROM\_EEBLOCK\_R = i;

**while**(EEPROM\_EEDONE\_R & EEPROM\_EEDONE\_WORKING);

}

//if there is a possibility of a matching password, increment to next word index

**if**(possible)

{

w\_index++;

}

//if no possibility is present, start at the beginning with new sequence.

**else**

{

buffer[0] = 0;

w\_index = 1;

}

}

//After analyzing the signal, return to edge mode to detect the next button press from IR remote

edgeMode();

}

TIMER1\_ICR\_R |= 1; //clear timeout interrupt in timer1

}

**char** **intToChar**(**int** number)

{

**if**(number >= 0 && number <= 9)

{

**return** number+48;

}

**return** 0;

}

**int** **numberSize**(**int** number)

{

**int** count = 0;

**if**(number == 0)

count++;

**while**(number)

{

number = number/10;

count++;

}

**return** count;

}

**void** **intToStr**(**char**\* str, **int** number)

{

**int** digit = 0;

**int** index = numberSize(number);

str[index] = 0;

**while**(index)

{

index--;

digit = number%10;

str[index] = intToChar(digit);

number = number/10;

}

}

**int** **strToInt**(**char**\* number)

{

**int** sum = 0;

**int** index = strlen(number)-1; //start from right to left(LSB)

**int** placeholder = 1; // start from one's place

**while**(number[index] && index >= 0)

{

**if**(number[index] < '0' || number[index] > '9') //if an invalid character is entered, return error

{

**return** -1;

}

sum += (number[index]-48)\*placeholder; // add to sum.

placeholder = placeholder\*10; // shift to left in significance.

index--;

}

**return** sum;

}

//Convert strings of hours and minutes to seconds in int data type

**int** **timeToSecs**(**char**\* hrs, **char**\* mins)

{

**int** sum = 0;

**int** index = strlen(hrs)-1; // get the LSB index

**if**(index > 1)

{

**return** -1;

}

**int** placeholder=1;

**while**(hrs[index] && index >= 0)

{

**if**(hrs[index] < '0' || hrs[index] > '9')// check if character is a valid number

{

**return** -1;

}

sum += (hrs[index]-48)\*3600\*placeholder; //convert from ascii value to int and add to sum of seconds

index--;

placeholder = placeholder\*10;

}

index = strlen(mins)-1; // get the LSB index

**if**(index > 1)

{

**return** -1;

}

placeholder = 1;

**while**(mins[index] && index >= 0)

{

**if**(mins[index] < '0' || mins[index] > '9') //Check if character is a valid number

{

**return** -1;

}

sum += (mins[index]-48)\*placeholder\*60; //convert from ascii value to seconds from minutes and add to sum

index--;

placeholder = placeholder\*10;

}

**if**(sum > 86400)

{

**return** -1;

}

**return** sum;

}

**void** **displayBlock**(uint8\_t P)

{

//Print contents of block P

EEPROM\_EEBLOCK\_R = P;

putsUart0("\n\r");

**int** i=0;

**while**(i < 16)

{

EEPROM\_EEOFFSET\_R = i;

**int** number = EEPROM\_EERDWR\_R;

**char** num[3];

intToStr(num,number);

putsUart0(num);

putcUart0(' ');

i++;

}

}

**void** **clearBlock**(uint8\_t P)

{

EEPROM\_EEBLOCK\_R = P;

EEPROM\_EEOFFSET\_R = 0;

**int** i=0;

**while**(i < 16)

{

EEPROM\_EEOFFSET\_R = i;

EEPROM\_EERDWR\_R = 0;

i = i+1;

**while**(EEPROM\_EEDONE\_R & EEPROM\_EEDONE\_WORKING);

}

}

**void** **setPassword**(uint8\_t P)

{

clearBlock(P); //Clear password block set up for password recording.

password\_length = 0; //Contains the length of password of current block/Contains default max password length

w\_index = 1; //EEPROM index for storing button codes

putsUart0("\n\rSetting password ..");

setPass = 1; //Enable password-setting

waitMicrosecond(10000000); //Set password values for 10 seconds

setPass = 0; //Disable password-setting

EEPROM\_EEOFFSET\_R = 1; //go to word 1, in block P to see if any button was pressed

**if**(EEPROM\_EERDWR\_R == 0) //if nothing is pressed, cancel and leave block empty.

{

putsUart0("Cancelled.");

w\_index = 1; //reset default value for w\_index;

password\_length = 15; //reset the maximum value as default password\_length

**return**;

}

EEPROM\_EEOFFSET\_R = 0; //get ready to store password length at word 0

EEPROM\_EERDWR\_R = password\_length;

**while**(EEPROM\_EEDONE\_R & EEPROM\_EEDONE\_WORKING);

putsUart0("Password set."); //else, set password length at word 0.

w\_index = 1; //reset default value for w\_index;

password\_length = 15; //reset the maximum value as default password\_length

}

**displayTime**(**int** secs)

{

//Printing Time

**int** buffer = 0;

**int** hrs = secs/3600; // convert to 24 hour scale

buffer = hrs\*3600; // get complete hours from total seconds

secs -= buffer; // remove hours from total seconds

**int** mins = secs/60; // convert remaining to minutes

buffer = mins\*60; // get total complete minutes

secs -= buffer; // remove minutes from seconds

// remainder is seconds

**char** hours[3];

**char** minutes[3];

**char** seconds[3];

intToStr(seconds, secs);

intToStr(minutes, mins);

intToStr(hours, hrs);

print("");

putsUart0("Hours\t");

print(hours);

putsUart0("Minutes\t");

print(minutes);

putsUart0("Seconds\t");

putsUart0(seconds);

}

**void** **buttonPress**()

{

//unlock/lock with physical button on board, unless it is "panicking" and local is enabled.

**if**(local && !panic)

{

beep();

**if**(!SENSOR)

{

unlock();

}

**else** **if**(SENSOR)

{

lock();

}

}

GPIO\_PORTF\_ICR\_R |= BUTTON\_MASK;

}

//prints parsed words as a debugging tool

**void** **printWords**(**char**\*\* words, uint8\_t maxWords)

{

**int** i=0;

**while**(i < maxWords)

{

**if**(words[i] == NULL)

**return**;

**char** index[255];

intToStr(index, i);

putsUart0(index);

putsUart0(" ");

putsUart0(words[i]);

i++;

}

putsUart0("\n\r");

}

**void** **execute\_cmd**(**char**\*\* words, uint8\_t arg\_count)

{

beep();

**if**(strcmp(words[0], "lock") == 0 && arg\_count == 1)

{

lock();

}

**else** **if**(strcmp(words[0], "unlock") == 0 && arg\_count == 1)

{

unlock();

}

**else** **if**(strcmp(words[0], "status") == 0 && arg\_count == 1)

{

putsUart0("\n\r");

**if**(status == 1)

{

putsUart0("Locked");

}

**else** **if**(status == 0)

{

putsUart0("Unlocked");

}

**else** **if**(status == -1)

{

putsUart0("Jammed");

}

**else**

{

putsUart0("ERROR: Unknown status");

}

}

**else** **if**(strcmp(words[0],"password") == 0 && arg\_count == 2)

{

**int** P = strToInt(words[1]);

//check if input password index is not a number

**if**(P == -1)

{

putsUart0("\n\r");

putsUart0("ERROR: Argument must be valid number(0-30).");

}

//Password must be between the range 0-30

**else** **if**(P < 0 && P > 30)

{

putsUart0("\n\r");

putsUart0("ERROR: Argument is not within valid range(0-30).");

}

setPassword(P);

//displayBlock(P);

}

**else** **if**(strcmp(words[0],"time") == 0 && arg\_count == 3)

{

**int** time = timeToSecs(words[1],words[2]);

**if**(time == -1)

{

putsUart0("\n\rError: Invalid time value");

**return**;

}

**while**(!(HIB\_CTL\_R & HIB\_CTL\_WRC));

HIB\_RTCLD\_R = time;

}

**else** **if**(strcmp(words[0],"autolock") == 0 && arg\_count >= 2 && arg\_count <= 3)

{

**if**(strcmp(words[1], "disable") == 0 && arg\_count == 2)

{

**while**(!(HIB\_CTL\_R & HIB\_CTL\_WRC));

HIB\_IM\_R &= ~HIB\_IM\_RTCALT0; //turn off interrupts

}

**else** **if**(strcmp(words[1],"disable") == 0 && arg\_count == 3)

{

putsUart0("\n\rError: Invalid argument");

}

**else**

{

//clear any previous interrupts

**while**(!(HIB\_CTL\_R & HIB\_CTL\_WRC));

HIB\_IC\_R |= 1;

//turn on interrupts for matching times

**while**(!(HIB\_CTL\_R & HIB\_CTL\_WRC));

HIB\_IM\_R |= HIB\_IM\_WC | HIB\_IM\_RTCALT0;

//convert hours and minutes to seconds to be put into RTCM0 register as the matching value

**int** time = timeToSecs(words[1], words[2]);

**if**(time == -1)

{

putsUart0("\n\rError: Invalid time value");

**return**;

}

**while**(!(HIB\_CTL\_R & HIB\_CTL\_WRC));

HIB\_RTCM0\_R = time;

}

}

**else** **if**(strcmp(words[0],"gettime") == 0 && arg\_count == 1)

{

**int** time = HIB\_RTCC\_R;

displayTime(time);

}

**else** **if**(strcmp(words[0],"beeper") == 0 && arg\_count == 2)

{

**if**(strcmp(words[1],"on") == 0)

{

beepEnable = 1;

}

**else** **if**(strcmp(words[1],"off") == 0)

{

beepEnable = 0;

}

**else**

{

**goto** ERROR;

}

}

**else** **if**(strcmp(words[0],"local") == 0 && arg\_count == 2)

{

**if**(strcmp(words[1],"on") == 0)

{

local = 1;

}

**else** **if**(strcmp(words[1],"off") == 0)

{

local = 0;

}

**else**

{

**goto** ERROR;

}

}

**else** **if**(strcmp(words[0],"led") == 0 && arg\_count == 2)

{

**if**(strcmp(words[1],"on") == 0)

{

**if**(SENSOR)

{

RED\_LED = 0;

GREEN\_LED = 1;

}

**else**

{

RED\_LED = 1;

GREEN\_LED = 0;

}

}

**else** **if**(strcmp(words[1],"off") == 0)

{

RED\_LED = 0;

GREEN\_LED = 0;

}

**else**

{

**goto** ERROR;

}

}

**else** **if**(strcmp(words[0],"panic") == 0 && arg\_count == 1)

{

setPassword(31);

}

**else** **if** (strcmp(words[0],"erase") == 0 && arg\_count == 2)

{

**int** P = strToInt(words[1]);

//check if input password index is not a number

**if**(P == -1)

{

putsUart0("\n\r");

putsUart0("ERROR: Argument must be valid number(0-30).");

}

//Password must be between the range 0-30

**else** **if**(P < 0 && P > 31)

{

putsUart0("\n\r");

putsUart0("ERROR: Argument is not within valid range(0-30).");

}

clearBlock(P);

displayBlock(P);

}

**else** **if**(strcmp(words[0],"clear") == 0 && arg\_count == 1)

{

**int** block = 0;

**while**(block < 32)

{

clearBlock(block);

block++;

}

putsUart0("\n\rAll passwords cleared");

}

**else**

{

ERROR:

putsUart0("\n\rINVALID COMMAND");

}

}

**void** **wc**(**void**)

{

//Checks if RTC was interrupted by match in RTCM0. AKA matching times

**if**(HIB\_RIS\_R & HIB\_RIS\_RTCALT0)

{

**if**(!SENSOR)

{

unlock();

}

**else**

{

lock();

}

}

**else**

{

**if**(HIB\_RTCC\_R >= 86400) //reset if it goes over 24 hours worth of seconds

{

**while**(!(HIB\_CTL\_R & HIB\_CTL\_WRC));

HIB\_RTCLD\_R = 0;

}

}

//clear both wrc and time match interrupt.

**while**(!(HIB\_CTL\_R & HIB\_CTL\_WRC));

HIB\_IC\_R |= 0x11;

}

/\*\*

\* main.c

\*/

**int** **main**(**void**)

{

initHw();

edgeMode();

beepEnable = 1;

beep();

//cmd line code

**const** uint8\_t max\_chars = 100;

**char** str[max\_chars];

**char**\* words[4];

uint8\_t num\_words=0;

**while**(1)

{

//Command line output

putsUart0("\r\n> ");

str[0] = '\0';

getsUart0(str, max\_chars); // get new cmd

num\_words = parseStr(str, words, max\_chars, 4); //parse the words up to 4 words due to max number of arguments

//printWords(words, num\_words);

execute\_cmd(words, num\_words);

}

**return** 0;

}