```
/*      Authors: Ricardo Goto and Craig Ross
        Project: RFID Security system
*/

#include <Mega32.h>
#include <stdio.h>
#include <delay.h>

/* Define Constants */
#define NUM_BITS 1080
#define NUM_CODES 20
#define RED 0b11111110
#define GREEN 0b11111101
#define OFF 0b11111111
#define CHECK_RECEIVE_TIME 50
#define NORMAL 0
#define REMOTEPOS 1
#define REMOTEAMT 2

/* Define input pin from RFID circuit */
#define RFIDIN PINA.0
#define LED PORTC


/* Prototypes */
void initialize(void);
void gets_str(void);
void puts_str(void);
void find_start_code(void);
void reduce_sequence(void);
void manchester_decode(void);
unsigned char match_code(unsigned char *char1,unsigned char *char2);
void copy_code(unsigned char *char1,unsigned char *char2);
unsigned char verify_code(unsigned char char1[]);
void check_receive(void);
void time_counter(void);
void store_into_bank(char position, char code[], char offset);

/* Global Variables */
char curr_sample;
char code_count;
char start_flag;
int totalBits;
char bit_array[NUM_BITS+1];
int sample_buffer;
int start_sequence;
int end_sequence;
char reduced_array[91];
char final_code[46];
char code_check[46];
char check_receive_timer;
int door_timer;
char mode;
signed char add_pos;
char del_pos;
char add_amt;
unsigned int seconds;
```

```
unsigned int minutes;
unsigned int hours;
unsigned int days;
unsigned char r_days[4];
unsigned char r_hours[3];
unsigned char r_minutes[3];
char no_code[46];
eeprom char code_bank[NUM_CODES][46];
char temp[46];
eeprom char bank_status[NUM_CODES];
char r_index, r_buffer[128], r_ready, r_char;
char t_index, t_buffer[128], t_ready, t_char;

/* USART recieve interrupt */
interrupt[USART_RXC] void uartr(void)
{
    r_char = UDR;
    UDR = r_char;
    if (r_char == 8)
    {
        if(r_index != 0) r_index--;
    }
    else if (r_char != '\r') r_buffer[r_index++] = r_char;
    else
    {
            r_buffer[r_index] = 0x00;
        r_ready = 1;
        UCSRB.7 = 0;
        putchar('\n');
    }
}

/* USART transmit interrupt */
interrupt[USART_DRE] void uartt(void)
{
    t_char = t_buffer[++t_index];
    if (t_char == 0)
    {
        UCSRB.5 = 0;
        t_ready = 1;
    }
    else UDR = t_char;
}

/* Timer0 compare match ISR */
interrupt [TIM0_COMP] void timer0_compare(void)
{
       /* Decrement timers every millisecond */
       if(check_receive_timer > 0) --check_receive_timer;
       if(door_timer > 0) door_timer--;
       if (seconds > 0) --seconds;
}

/* External pin interrupt */
interrupt[EXT_INT2] void int2(void)
{
    /* In this interrupt, we want to sample the output of the
```

```
       second DFF just after the rising edge of the output of
       the comparator stage. This will tell us how the length
       of the pulse corresponds to the number of bits. We only
       start caring about the data when it first goes high,
       however, the card will probably be close to the
       maximum reading distance. Therefore, we wait until the
       card gets closer (we assume the person is bringing the
       card to about 1-2 inches from the reader coil) and then
       we start to sample. Once we have enough samples to
       capture a full period of the looping response, we stop
       sampling.
    */
    delay_us(16);
    curr_sample= RFIDIN;
    if((start_flag == 0) && (curr_sample == 1))
    {
        start_flag= 1;
        sample_buffer++;
    }
    else if((start_flag == 1) && (sample_buffer <= 500))
sample_buffer++;
    else if((start_flag == 1) && (totalBits < NUM_BITS))
    {
        if(curr_sample == 1) bit_array[totalBits++]= 49;
        else bit_array[totalBits++]= 48;
    }
}


void main(void)
{
    char i;
        initialize();
        while(1)
        {
            if((totalBits == NUM_BITS) && (door_timer == 0) && (mode ==
NORMAL))
            {
            /* Normal Operation:
               When a full period of the looping card response is
                   captured, we want to decode the response. We execute
                   the following steps until we get 3 identical codes
                   in a row.
                   - turn off the external pin interrupt since we are
                     not going to be reading anything in this window of
                     time
                   - look for a start code and take only data
                   - reduce the bit sequence to 90 bits
                   - manchester decode to 45 bits
                   If we get 3 identical codes, then we compare that
code
                   with the code bank to see if the code is currently
                   allowed access to the facility. If so, we blink a
green
                   LED (open the door) for 3 seconds. If not, then we
                   blink a red LED (do not open the door) which also
lasts
```

```
                   for 3 seconds.
                */
                GICR= 0b00000000;
                find_start_code();
                reduce_sequence();
                manchester_decode();
            if(code_count == 0)
            {
                copy_code(code_check,final_code);
                code_count= 1;
            }
            else if(code_count == 1)
            {
                if(match_code(code_check,final_code)) code_count= 2;
                else code_count= 0;
            }
            else if(code_count == 2)
            {
                if(match_code(code_check,final_code))
                {
                    if(verify_code(final_code))
                    {
                        sprintf(t_buffer,"\r\nCard ID: %s\r\nAccess
Granted at %03d:%02d:%02d\r\n\r\n",final_code,days,hours,minutes);
                        puts_str();
                        while(t_ready == 0){};
                        LED= GREEN;
                        door_timer= 3000;
                    }
                    else
                    {
                        sprintf(t_buffer,"\r\nCard ID: %s\r\nAccess
Denied at %03d:%02d:%02d\r\n\r\n",final_code,days,hours,minutes);
                        puts_str();
                        while(t_ready == 0){};
                        LED= RED;
                        door_timer= 3000;
                    }
                }
                code_count= 0;
            }
            /* Make sure to reset the interrupt variables
               and turn on the external interrupt.
            */
            totalBits= 0;
                start_flag= 0;
                sample_buffer= 0;
                GICR= 0b00100000;
            }
            else if(mode == REMOTEAMT)
            {
            /* Remote Quantity Add Operation:
               If we are in this mode of operation, this segment of
code
               makes it so that we loop in Remote Positional Add until
               the amount of codes have been added. Our algorithm
searches
```

```
               the code bank status to find open positions and uses
Remote
               Positional Add to add the card ID.
            */
            add_pos= -1;
            for(i= 0; i<NUM_CODES; i++)
                {
                    if(bank_status[i]== 0)
                    {
                        add_pos= i;
                        bank_status[i]= 1;
                        break;
                    }
                }
                if(add_pos != -1)
                {
                 mode= REMOTEPOS;
                    printf("Please place card in front of reader\r\n");
                    delay_ms(500);
                    LED= GREEN & RED;
                    delay_ms(200);
                    LED= OFF;
                    delay_ms(300);
                    LED= GREEN & RED;
                    delay_ms(200);
                    LED= OFF;
                    add_amt--;
                }
                else
                {
                    printf("Code bank is full, please delete unwanted
codes and re-run\r\n");
                    add_amt= 0;
                    mode= NORMAL;
                }
            }
            else if((totalBits == NUM_BITS) && (mode == REMOTEPOS))
            {
                /* Remote Positional Add Operation:
                   Similar to normal mode except we are storing the
                   next card into the code bank. To prevent a bad code
                   from getting into the code bank, we add even more
                   error protection by having to read the same code 5
                   times in a row before it is accepted into the code
                   bank.
                */
                GICR= 0b00000000;
                find_start_code();
                reduce_sequence();
                manchester_decode();
                if(code_count < 5)
                {
                    if(code_count == 0)
                    {
                        copy_code(code_check,final_code);
                        code_count++;
                    }
```

```
                    else
                    {
                        if(match_code(code_check,final_code))
code_count++;
                    else code_count= 0;
                    }
                }
                if(code_count == 5)
                {
                    store_into_bank(add_pos,final_code,0);
                    sprintf(t_buffer,"Code added to position
%d\r\n",add_pos);
                    puts_str();
                    LED= GREEN;
                    delay_ms(200);
                    LED= RED;
                    delay_ms(200);
                    LED= GREEN & RED;
                    delay_ms(500);
                    LED= OFF;
                    code_count= 0;
                    if(add_amt > 0)mode= REMOTEAMT;
                    else mode= NORMAL;
                }
                totalBits= 0;
                start_flag= 0;
                sample_buffer= 0;
                GICR= 0b00100000;
            }

            /* Check timers for timed tasks */
            if(check_receive_timer == 0) check_receive();
            if((door_timer == 0) && ((LED == GREEN) || (LED == RED)))
LED= OFF;
            if(seconds == 0) time_counter();
        }
}

/* Function: Check Receive
   This function checks the r_ready flag every 50 milliseconds. If
   there is a requested command in the r_buffer then we process the
   command and execute the corresponding code.
*/
void check_receive(void)
{
    char i,j;
    check_receive_timer= CHECK_RECEIVE_TIME;
    if(r_ready == 1)
    {
        /* Add input code at specific position */
        if(r_buffer[0] == 'a')
        {
            if(r_buffer[3] == ' ')
            {
                add_pos= r_buffer[2]-48;
                if((add_pos >= 0) && (add_pos <= 9))
                {
```

```
            store_into_bank(add_pos,r_buffer,4);
            bank_status[add_pos]= 1;
            printf("Code added at position %d\r\n",add_pos);
        }
        else printf("Invalid code position\r\n");
    }
    else if(r_buffer[4] == ' ')
    {
        add_pos= (r_buffer[2]-48)*10+(r_buffer[3]-48);
        if((add_pos >=0) && (add_pos < NUM_CODES))
        {
            store_into_bank(add_pos,r_buffer,4);
            bank_status[add_pos]= 1;
            printf("Code added at position %d\r\n",add_pos);
        }
        else printf("Invalid code position\r\n");
    }
    else printf("Incorrect syntax\r\n");
}
/* List all codes and corresponding status */
else if(r_buffer[0] == 'l')
{
    putsf("Code Bank:\r\n");
    for(i= 0; i < NUM_CODES; i++)
    {
        for(j= 0; j < 45; j++)
        {
            temp[j]= code_bank[i][j];
        }
        printf("%02d: %d - %s\r\n",i,bank_status[i],temp);
    }
}
/* Delete code at specific position */
else if(r_buffer[0] == 'd')
{
    if(r_buffer[3] == '\0')
    {
        del_pos= r_buffer[2]-48;
        if((del_pos >= 0) && (del_pos <= 9))
        {
            store_into_bank(del_pos,no_code,0);
            bank_status[del_pos]= 0;
            printf("Code deleted at position %d\r\n",del_pos);
        }
        else printf("Invalid code position\r\n");
    }
    else if(r_buffer[4] == '\0')
    {
        del_pos= (r_buffer[2]-48)*10+(r_buffer[3]-48);
        if((del_pos >=0) && (del_pos < NUM_CODES))
        {
            store_into_bank(del_pos,no_code,0);
            bank_status[del_pos]= 0;
            printf("Code deleted at position %d\r\n",del_pos);
        }
        else printf("Invalid code position\r\n");
    }
```

```
    else printf("Incorrect syntax\r\n");
}
/* Unlock door for 3 seconds */
else if(r_buffer[0] == 'u')
{
    LED= GREEN;
    door_timer= 3000;
}
/* Enable remote positional add mode */
else if((r_buffer[0] == 'r') && (r_buffer[1] == 'p'))
{

    if(r_buffer[4] == '\0')
    {
        add_pos= r_buffer[3]-48;
        if((add_pos >= 0) && (add_pos <= 9))
        {
            bank_status[add_pos]= 1;
            printf("Please place card in front of reader\r\n");
            LED= GREEN & RED;
                delay_ms(400);
                LED= OFF;
                delay_ms(400);
                LED= GREEN & RED;
                delay_ms(400);
                LED= OFF;
            mode= REMOTEPOS;
        }
        else printf("Invalid code position\r\n");
    }
    else if(r_buffer[5] == '\0')
    {
        add_pos= (r_buffer[3]-48)*10+(r_buffer[4]-48);
        if((add_pos >=0) && (add_pos < NUM_CODES))
        {
            bank_status[add_pos]= 1;
            printf("Please place card in front of reader\r\n");
            LED= GREEN & RED;
                delay_ms(400);
                LED= OFF;
                delay_ms(400);
                LED= GREEN & RED;
                delay_ms(400);
                LED= OFF;
            mode= REMOTEPOS;
        }
        else printf("Invalid code position\r\n");
    }
    else printf("Incorrect syntax\r\n");
}
/* Enable remote quantity add mode */
else if((r_buffer[0] == 'r') && (r_buffer[1] == 'a'))
{
    if(r_buffer[4] == '\0')
    {
        add_amt= r_buffer[3]-48;
        if((add_amt >= 1) && (add_amt <= 9)) mode= REMOTEAMT;
```

```
                else printf("Invalid add quantity\r\n");
            }
            else if(r_buffer[5] == '\0')
            {
                add_amt= (r_buffer[3]-48)*10+(r_buffer[4]-48);
                if((add_amt >=1) && (add_amt <= NUM_CODES)) mode=
REMOTEAMT;
                else printf("Invalid add quantity\r\n");
            }
            else printf("Incorrect syntax\r\n");
        }
        /* Help */
        else if(r_buffer[0] == 'h')
        {
            putsf("Commands:\r");
            putsf("a  - add a code -                 syntax: a <pos>
<code>\r");
            putsf("l  - list all codes -             syntax: l\r");
            putsf("d  - delete a code -              syntax: d
<pos>\r");
            putsf("u  - unlock door -                syntax: u\r");
            putsf("rp - remote add (positional) -    syntax: rp
<pos>\r");
            putsf("ra - remote add (amount) -        syntax: ra
<amt>\r");
        }
        /* Unrecognized commands */
        else printf("Unrecognized Command\r\n");
        gets_str();
    }
}

/* Function: Time Counter
   This function keeps track of time for reporting to
   the hyperterm.
*/
void time_counter(void)
{
    seconds= 59999;
    if(minutes < 59) minutes++;
    else
    {
        minutes= 0;
        if(hours < 23) hours++;
        else
        {
            hours= 0;
            if(days < 364) days++;
            else days= 0;
        }
    }
}

/* Function: Store Into Bank
   Stores a code into code bank at a specific position starting with
   a specific offset of code.
*/
```

```
void store_into_bank(char position, char code[], char offset)
{
    unsigned char i;
    for(i= 0; i < 45; i++) code_bank[position][i]= code[i+offset];
}

/* Function: Verify Code
   This function takes a code for an argument and checks
   to see if the code matches with any of the codes in
   the code bank. Returns a 1 if there is a match and a 0
   otherwise.
*/
unsigned char verify_code(unsigned char char1[])
{
    unsigned char i= 0;
    unsigned char j= 0;
    unsigned char code_match= 0;
    unsigned char element_match;
    while((code_match == 0) && (i < NUM_CODES))
    {
        if(bank_status[i]==1)
        {
            element_match= 1;
            while((element_match == 1) && (j <45))
            {
                if(code_bank[i][j] != char1[j]) element_match= 0;
                j++;
            }
            code_match= element_match;
        }
        i++;
    }
    return code_match;
}

/* Function: Match Code
   This function compares two codes by comparing the
   elements of each array. Returns a 1 if there if
   there is a match and a 0 otherwise.
*/
unsigned char match_code(unsigned char char1[], unsigned char char2[])
{
    char i= 0;
    char correct= 1;
    while((correct == 1) && (i < 45))
    {
        if(char1[i] != char2[i]) correct= 0;
        i++;
    }
    return correct;
}

/* Function: Copy Code
   Copies a code from source char2 to destination char1.
*/
void copy_code(unsigned char char1[], unsigned char char2[])
{
```

```
    unsigned char i;
    for(i= 0; i < 45; i++) char1[i]= char2[i];
}

/* Function: Find Start Code
   To find the start code, we must look for a sequence of 15 to 18 1's.
   We run through the bit stream received from the card and look for
1's.
   Anytime there is a 0, we reset the counter. Eventually the counter
   will get to 15 or more and then we have detected the start sequence.
   Since we know there are 540 bits in a period of the card response,
we
   know the end of the sequence is 539 bits later.
*/
void find_start_code(void)
{
    int i= 0;
    char sequence= 0;
    char count= 0;
    start_sequence= 0;
    end_sequence= 0;
    while((i < totalBits))
    {
        if(bit_array[i] == 49)
        {
            sequence= 1;
            count++;
        }
        else
        {
            if((sequence == 1) && (count >= 15))
            {
                start_sequence= i-count;
                end_sequence= start_sequence+539;
                i= totalBits+1;
            }
            sequence= 0;
            count= 0;
        }
        i++;
    }
    if (i == totalBits) start_sequence= -1;
}

/* Function: Reduce Sequence
   Once we have the 540 bit card response, we need to remove
   the redundancy. The code looks something similar to:
   0000011111100000011111000000000001111110000011111...
   We recognize 5 to 6 bits as a single bit and a stream of
   10, 11, or 12 bits is two bits. Thus the code above would
   decode to 010100101. There are 90 bits in the reduced form.
   This function transforms the redundant bit stream to the
   reduced form by detecting sequences of 5/6 and 10/11/12
   and creating a new array.
*/
void reduce_sequence(void)
{
```

```
    int i;
    signed char j= -1;
    unsigned char count;
    unsigned char value;
    i= start_sequence;
    while(i <= end_sequence)
    {
        count= 0;
        value= bit_array[i];
        if(j == 89)
        {
            reduced_array[j]= value;
            i= end_sequence+1;
        }
        else
        {
            while(value == bit_array[i])
            {
                count++;
                i++;
            }
            if(j == -1) j++;
            else if((count == 5) || (count ==6)) reduced_array[j++]=
value;
            else if((count == 10) || (count == 11) || (count == 12))
            {
                reduced_array[j++]= value;
                reduced_array[j++]= value;
            }
        }
    }
}
/* Function: Manchester Decoder
   This function translates the reduced bit stream that
   is Manchester coded to raw data. This final bit stream
   is the code we used to identify different cards.
*/
void manchester_decode(void)
{
    unsigned char i;
    unsigned char j= 0;
    for(i= 0; i< 90; i+=2)
    {
        if((reduced_array[i] == 49) && (reduced_array[i+1] == 48))
final_code[j++]= 48;
        else final_code[j++]= 49;
    }
}

void initialize(void)
{
    char i;
    /* PORTD.7 generates the 125kHz square wave using timer2
       output compare.
       - PORTD is set for output
       - Waveform Generation Mode is set to "CTC"
       - Compare Output Mode is set to "Toggle 0C2 on compare match"
```

```
       - Output Compare Register is set to 63
       - Clock Select is set to "No prescaling"
       Timer2 will increment at clock speed (16 MHz) to 63 and then
reset.
       Once it reaches this value, it will toggle OC2 to generate the
0V-5V
       125kHz square wave.
    */
    DDRD= 0xff;
    TCCR2= 0b00011001;
        OCR2= 63;

        /* RFIDIN is PORTA.0 */
        DDRA= 0x00;

        /* Enable LED port as output */
        DDRC= 0xff;
        LED= OFF;

        /* Setup Timer0
           - Turn on timer0 ISR
           - Waveform Generation Mode is set to "CTC"
           - Output Compare Register is set to 249
           - Clock Select is set to "Clock/64"
           - Initialize the counter to 0
           Timer0 interrupt occurs every millisecond.
        */
        TIMSK= 0b00000010;
        TCCR0= 0b00001011;
        OCR0= 249;
        TCNT0= 0;


        /* Setup the USART
           - Enable the Transmitter and Receiver
           - Set the Baud Rate to 9600Hz
        */
    UCSRB = 0x18;
    UBRRL = 103;

    /* Initialize variables */
    start_flag= 0;
    curr_sample= 0;
    totalBits= 0;
    sample_buffer= 0;
    code_count= 0;
    check_receive_timer= CHECK_RECEIVE_TIME;
    mode= NORMAL;

    /* Initialize Code Bank Status */

    for(i= 0; i < NUM_CODES; i++)
    {
        if (bank_status[i] == "") bank_status[i]= 0;
    }

        /* Enable Interrupts */
```

```
        #asm
                sei
        #endasm

        /* External pin interrupt
           - Enable in GICR
           - Set for rising edge
           - Clear interrupt flag
           - PORTB as input
        */
        GICR= 0b00100000;
    MCUCSR= 0b01000000;
    DDRB.2= 0;

        /* Send a starting message to hypertrm */
        putsf("\r\nInitializing the Reader...\r\n");
        r_ready = 0;
        t_ready = 1;
        gets_str();

    /* Initialize the data and time by prompting admin */
        putsf("Please enter time in the format: DDD:HH:MM\r\n");

        while(r_ready == 0){};
    r_days[0] = r_buffer[0];
    r_days[1] = r_buffer[1];
    r_days[2] = r_buffer[2];
    r_days[3] = '\0';
    r_hours[0] = r_buffer[4];
    r_hours[1] = r_buffer[5];
    r_hours[2] = '\0';
    r_minutes[0] = r_buffer[7];
    r_minutes[1] = r_buffer[8];
    r_minutes[2] = '\0';
    sscanf(r_hours,"%d",&hours);
    sscanf(r_minutes,"%d",&minutes);
    sscanf(r_days,"%d",&days);
    gets_str();
    seconds = 59999;
    sprintf(t_buffer,"Current Date and Time:
%03d:%02d:%02d\r\n",days,hours,minutes);
    puts_str();
    while(t_ready == 0);

    putsf("Please type h for help\r");
}

/* Complete receiving from hypertrm */
void gets_str(void)
{
  r_ready = 0;
  r_index = 0;
  UCSRB.7 = 1;
}

/* Begin transmitting to hypertrm */
void puts_str(void)
```

```
{
  t_ready = 0;
  t_index = 0;
  putchar(t_buffer[0]);
  UCSRB.5 = 1;
}
```