LABORATORY REPORT

To: Dr.Randy Hoover

From: Benjamin Lebrun, Benjamin Garcia

Subject: Lab Assignment 3: UART Security

Date: March 15, 2019

Introduction

This lab required us to create a simulated 'security system' with three states: "locked", "unlocked", and "reset". While the system is in each state, it should light up an LED corresponding to the state, and prompt the user for input through the terminal based on the current state.

The microcontroller maintains a password value which is initialized to '1234' and can be updated by entering the 'reset' mode. The microcontroller begins in the "locked" state and can be "unlocked" with the password. A secondary requirement was that the user's inputs would be echoed back to them with each keypress.

Our implementation holds the current light configuration in register 19 (r19) which is aliased to 'light_config'. We also store information about the next expected input in register 18 (r18) aliased to 'input_mode'. Our program performs some initial configuration, before entering an infinite loop in main. From here, we use the receive complete interrupt to drive the program from user input.

The finished implementation starts with the LEDs off and then, after the setup code finishes, enters the 'locked' mode, and awaits user input over serial.

Equipment

This lab required the following equipment:

- 3 resistors (220Ω)
- 3 LEDs
- 4 wires (male-male)
- 1 USB-A to USB-B cable
- 1 breadboard
- 1 Arduino UNO (ATMega328P)
- Atmel Studio 7
- Putty

configuration

The Arduino was connected to the computer with the USB-A to USB-B cable. On the breadboard, each LED was placed in a circuit with a single resistor connected to ground. One of the Arduino's ground pins was connected to the breadboard's ground, and the LEDs were connected to PORTB pins 0, 1, and 2 (pins numbered 8, 9, 10 on the Arduino).

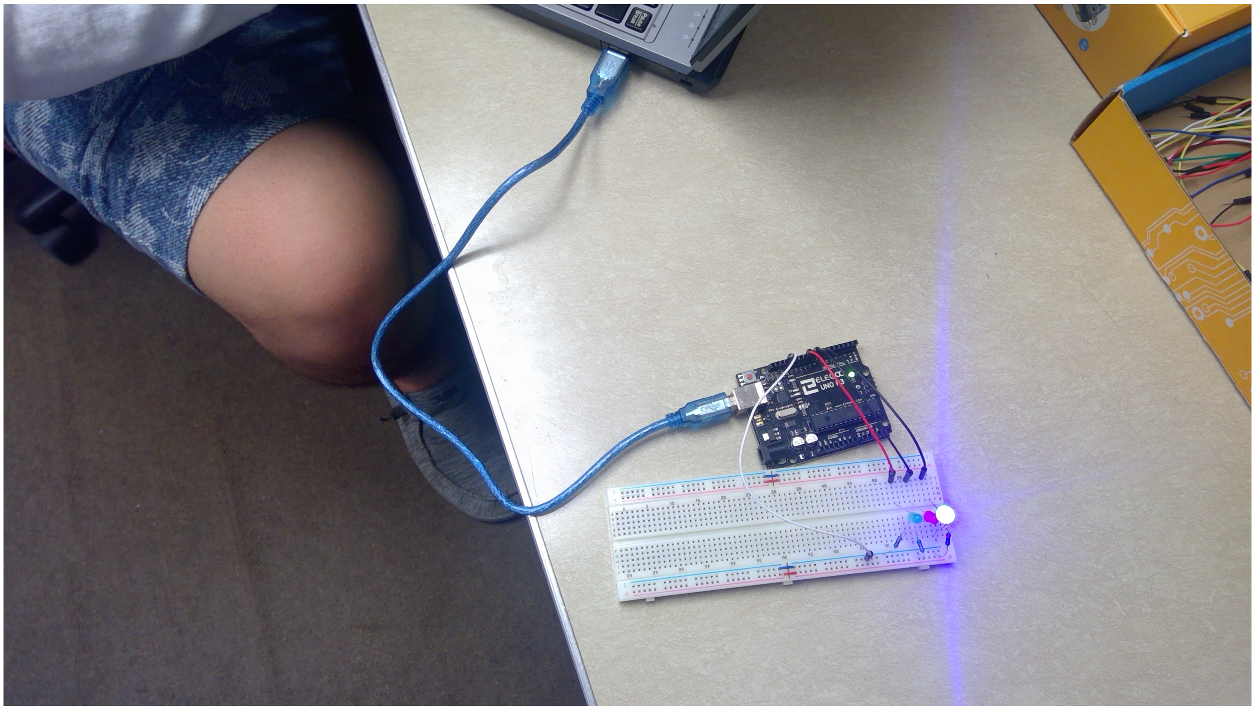

Figure 1: Picture of the wiring setup

Implementation

uart_rx (Interrupt)

uart_rx handles the UART receive complete interrupt. Once the data being read is available, the byte is sent to the handling function to determine what is to be done with it. After handling is complete, we return from the interrupt.

uart_init

uart_init is boilerplate from the previous lab, altered slightly to allow easier configuration of the interrupts. We presently use the UART receive complete interrupt to handle user input, so its enable bit is set here.

This function configures the UART control and status registers to allow writing and reading using the '8n1' protocol on the UART port. Additionally, communication is configured to use a baudrate of 9600.

uart_tx

Unlike the RX function, uart_tx is not an interrupt. uart_tx spin-locks until UDR0 is empty, and then sends the byte in serial_out.

print_string

print_string expects the address of the first character of the string to print to be in the Z pointer registers as a byte address. It loops through the characters in the string until a 0 (null-terminator) is reached. At this point the function will return.

print_menu

managing function to call print_string for each string in the menu. Initializes Z using the LSW macro for each string, then calls print_string.

LSW, Load String with Word address (macro)

to make filling the Z register easier for the print_string function, we created a macro named LSW that will fill the Z register from a label.

It doubles the value of the address input in order to get a byte address rather than a word address. This is necessary for LPM to work as expected.

set_lights

outputs the dedicated 'light_config' register to PORTB to set the LEDs on or off. Used by our state controlling convenience functions.

clear_lights

sets 'light_config' to 0x0 and then calls set_lights. Created to allow clearing the lights for debugging purposes.

set_locked

turns on the 'locked' light, turning off any other lights. Intended to make what is happening in the code clearer than simply assigning a raw value to 'light_config'.

set_unlocked

turns on the 'unlocked' light, turning off any other lights. Intended to make what is happening in the code clearer than simply assigning a raw value to 'light_config'.

set_reset

turns on the 'reset' light, turning off any other lights. Intended to make what is happening in the code clearer than simply assigning a raw value to 'light_config'.

init_pwd

initializes the password in SRAM to the specified '1234' and calls set_locked.

start

contains startup logic such as setting the global interrupt enable bit, calling uart_init, and calling init_pwd. Jumps to main afterwards.

main

a forever loop. Performs no additional logic as the program is interrupt driven.

recieve_input and test_input

These two functions attempt to retrieve an input from the uart, then uses the input to activate the different modes on the device.

set_locked

Sets the device into a locked mode, will print a message to the screen and set the corresponding LED on the device

uart_rx_wait

A function that calls the UART's receive functionality in a blocking mode. This is used when we must disable interrupt driven UART for the entire device.

load_pw_buffer

Function that loads the password buffer from the serial line in. Used when we prompt the user for a password, then stores it into a buffer segment in code.

set_unlocked

Function for handling unlocking the device. Will first prompt the user for a password. If correct, will unlock the device and illuminate the unlock LED. If incorrect, the device will inform the user and remain locked.

set_reset

Function handler used for the reset functionality of the device. Like the unlock handler, will prompt the user for a password. If incorrect, will lock itself. If correct, will prompt the user for a new password, then store it as the current password, then resend the new password to output.

load_new_password

Function will attempt to load a password from the password buffer into the current password in memory.

clear_lights

Not used outside of debugging purposes, clears the LEDs on the device.

set_lights

Takes the value of the light configuration register and will write it to the LED port.

Discussion

One problem we faced initially was printing the menu to the terminal. After reading the documentation on LPM and seeing it took Z, we had assumed it wanted the word-based memory address rather than the byte-based memory address. This was thankfully an easy fix in LSW where we doubled the input address before taking the high and low bytes.

Another challenge was figuring out a convenient way to store the state of the system (i.e. what the next input should be). We settled on storing the state in a dedicated register whose value would indicate what the next input is expected to be (and therefore which handling function should receive the input).

Responses

1.) What is the difference(s) between direct and indirect memory access? When is one used instead of the other?

Direct memory access is where an instruction contains information about the address or register where the value used in the instruction is stored.

Indirect memory access is where the address or register specified doesn't contain the value to use, but rather the address of the value to use.

Direct memory access is usually faster than Indirect access as the location of the value to use is known as part of the instruction itself. For this reason it is the most common choice, however if the address that is part of the instruction refers to a region of memory that cannot be represented in the space provided, then the alternative is indirect access.

With Indirect access, the memory location or register specified is actually functioning as a pointer to the actual location of the data to use. This can allow accessing data at a greater distance in memory (i.e. there may only be a few bits for an address, but if the address pointed to points elsewhere, there would be a full byte (or more) of range).

This extra layer of indirection comes at the cost of speed as mentioned previously. It is necessary to first read the address to read from first, before the instruction can actually retrieve any data.

2.) Our ATMega328Ps have several special registers, among them the X, Y, and Z registers. What makes these registers different from, say, r16? Are there differences between the X, Y, and Z registers? If so, give an example.

The X, Y, and Z registers are pointer registers, that is, they are actually pairs of registers (XL, XH), (YL, YH), and (ZL, ZH) that can be used to address the entire program memory space.

These registers each have special characteristics as well. The Y and Z registers can be used with displacements and post/pre increments, whereas X cannot. Additionally, only Z can be used to index into flash memory (the code segment).

3.) As given in Lab 02, the `UART_RX` and `UART_TX` functions had the same critical flaw pertaining to stack usage. What is this flaw, and what is the immediate consequence on return? Could this same flaw lead to a stack overflow?

The spin-lock used with UART_TX and UART_RX jumped to the beginning of the function, where the value of uart_buf would be pushed to the stack again. This meant that if UART_TX had to wait even one cycle, the top of the stack would change, and when ret was called at the end the program would use the value in uart_buf as part of the return address.

If UART_TX ended up waiting for an extended period, the stack would eventually reach the end of the general purpose SRAM and pushed values would be output to the reserved memory segment.

4.) Our ATMega328Ps have 1k of EEPROM memory. What is EEPROM memory, and why might it be useful to our microcontroller?

EEPROM stands for Electronically Erasable Programmable Read-Only Memory. Its main benefit over SRAM is the fact that is non-volatile and will retain its set value between resets. This makes it ideal for storing data that cannot be allowed to be lost.

For example, we could set our password in EEPROM, that way it will not be lost between resets.

5.) Endian-ness plays a large role in how information is stored in computing systems. Is the ATMega328P big-endian, or little-endian?

The ATMega328P is little-endian.

Appendices

The following files are included as appendices:

- main.asm - contains functions and interrupt handlers
- configuration.asm - contains constants, defines, aliases, and other configuration code
- macros.asm - contains macro definitions
- ui.asm - contains functions for simpler printing of status messages
- strings.asm - constants file for our different status messages

Appendix A: main.asm

```
;
; Project3.asm
;
; This project consists of a 'security system' where a user
; may specify a four digit password and lock or unlock the
; system. The password resets to '1234' when the chip is
; reset.
;
; The user may interact with the device over the serial port,
; and the user's inputs will be echoed back to them.
;
; The project demonstrates the use of arrays, multiple forms
; of memory access, and serial communication.
;
; Created: 3/12/2019 7:28:39 AM
; Authors : Benjamin LeBrun, Benjamin Garcia
;
;
; USING:
; PORTB Pins 0-7
;
; DDRB - The Port D Data Direction Register - read/write
; PORTB - The Port D Data Register - read/write
; PINB - The Port D Input Pins Register - read only
;
; PORTB = B10101000; // sets digital pins 7,5,3 HIGH
;


; data segment memory allocations
.dseg
; start at the first non-reserved address in SRAM
.org 0x0100
curr_pwd: .byte 4 ; reserve four bytes for the four digit password
pwd_buff: .byte 4 ; reserve four bytes for password buffer

; code segment start
.cseg
.org     0x0000
rjmp     start ; jump to start to get past interrupt code

; interrupt vectors
.org     0x0024 ; USART RX complete interrupt
rcall recieve_input
reti
.org     0x0026 ; USART Data Register Empty interrupt
;TODO
.org     0x0028 ; USART TX complete interrupt
;TODO

; interrupt handling code starts here
.org 0x0040

; void uart_init();
; initializes the UART for serial communication.
; enables the recieve and transmit complete interrupts,
; sets the baudrate, and finally enables communication.
uart_init:
        push     uart_buf
        push     uart_buf2
        lds              uart_buf, UCSR0B
        ; enable rx,tx, and interrupts selected above
        ; TODO: reformat (or change logic?) to reduce this below 80 chars
        ori              uart_buf, (RXC_EN<<RXCIE0)|(TXC_EN<<TXCIE0)|(UDRE_EN<<UDRIE0)|(1<<RXEN0)|(1<<TXEN0)
        sts              UCSR0B, uart_buf ; enables rx, tx, and interrupts

        lds              uart_buf, UCSR0C
        ori              uart_buf, 0x06
        sts              UCSR0C, uart_buf ; 8n1 async

        ldi              uart_buf, HIGH(BAUD_PRESCALE)
        ldi              uart_buf2, LOW(BAUD_PRESCALE)
        sts              UBRR0H, uart_buf
        sts              UBRR0L, uart_buf2
        pop              uart_buf2
        pop              uart_buf
        ret

; void uart_tx(serial_out);
; send the byte of information currently in the
; serial_out register.
uart_tx:
        push     uart_buf
        ; jump here while spinlocking to prevent running out of stack
        uart_tx_loop:
                lds              uart_buf, UCSR0A
                sbrs     uart_buf, UDRE0 ; Check if there's data to read
                rjmp     uart_tx_loop     ; Loop if there is waiting data
        sts              UDR0, serial_out ; write output
```

```
        pop             uart_buf
        ret

; char uart_rx();
; read the byte of information in UDR0
; into the serial_in register.
uart_rx:
        lds             serial_in , UDR0 ; read the data
        mov             serial_out , serial_in ; copy to serial_out
        rcall   uart_tx ; echo back input
        ret

.org 0x0100


; macro definitions
.include "macros.asm"
; configuration information , defines , equations , and aliases
.include "configuration.asm"
; include null-terminated string constants
.include "strings.asm"
; include ui functions
.include "ui.asm"

; startup code, called on every reset
start:
        clr             r1
        out             SREG, r1                        ; clear sreg for safety

        ldi             r16 , 0xff                      ; load 0x11111111 to reg 16
        out             DDRB, r16                       ; set portb to output

        clr             r16                                     ; clear it because it's one of our buffers

        ; Initialize the stack
        ldi             r28 , LOW(RAMEND)
        ldi             r29 , HIGH(RAMEND)
        out             SPL, r28
        out             SPH, r29

        ; initialize the security code to 0
        rcall   init_pwd

        ; set the lights , should be locked after init_pwd (red light)
        rcall   set_lights

        rcall   uart_init ; initialize the UART
        sei     ; global interrupt enable
    rjmp        main ; jump to main (in case something is between start and main)

; void main();
; main loop for the program.
main:
        LSW             clr_string
        rcall   print_string
        rcall   print_menu
        main_loop:
        rjmp    main_loop

; void init_pwd();
; initializes the password in the .dseg to '1234'.
init_pwd:
        .def    tmp = r16
        push    tmp
        ldi             tmp, '1'
        sts             curr_pwd , tmp
        ldi             tmp, '2'
        sts             curr_pwd + 1, tmp
        ldi             tmp, '3'
        sts             curr_pwd + 2, tmp
        ldi             tmp, '4'
        sts             curr_pwd + 3, tmp
        rcall   set_locked
        pop             tmp
        .undef  tmp
        ret

;void recieve_input(char* str);
;get input from user
recieve_input:
        rcall uart_rx
        mov input_mode, serial_in
        rcall test_input
        ret

; void uart_rx_wait()
; wait buffer for uart, wrapper around
; original uart_rx function that waits for
; buffer to fill , loads result into serial_in
uart_rx_wait:
        buffer_wait:
                lds             r16 , UCSR0A
```

```
                sbrs    r16, RXC0
                rjmp  buffer_wait

                rcall  uart_rx
        ret

; void load_pw_buffer()
; loads from UART the password buffer
load_pw_buffer:
        call    uart_rx_wait
        sts     pwd_buff, serial_in
        call    uart_rx_wait
        sts     pwd_buff + 1, serial_in
        call    uart_rx_wait
        sts     pwd_buff + 2, serial_in
        call    uart_rx_wait
        sts     pwd_buff + 3, serial_in
        ret

; void load_new_password();
; loads new password from uart_rx
load_new_password:
        rcall load_pw_buffer
        STSS curr_pwd, pwd_buff
        STSS curr_pwd+1, pwd_buff+1
        STSS curr_pwd+2, pwd_buff+2
        STSS curr_pwd+3, pwd_buff+3
ret
```

Appendix B: configuration.asm

```
;
; configuration.asm
;
; defines, equations, and other setup logic.
;
; Created: 3/12/2019 3:16:13 PM
;  Author: Benjamin Garcia, Benjamin LeBrun
;
;

; set clock frequency information and baudrate for serial communication
.equ    F_CPU = 16000000
.equ    BAUDRATE = 9600
.equ    BAUD_PRESCALE = F_CPU / (BAUDRATE*16)-1

; Interrupt enable defines; 0 = disable, 1 = enable
.equ    UDRE_EN = 0
.equ    RXC_EN = 1
.equ    TXC_EN = 0

; register aliases
.def     serial_in = r14
.def     serial_out = r15
.def     uart_buf = r16
.def     uart_buf2 = r17
.def     light_config = r19

; program status registers:
; these registers are used to store information about the program's state
.def     input_mode = r18 ; determines how to treat user input.
```

Appendix C: macros.asm

```
;
; macros.asm
;
; macros used in project3
;
;  Created: 3/12/2019 3:19:25 PM
;   Author: Benjamin Garcia, Benjamin LeBrun
;

; Load String with Word address (LSW)
; @1: label where the string starts
;
; load a label into the Z register.
; used to point Z at the start of the string to print
; in print_string.
.macro          LSW
                        ldi             ZL, LOW(@0<<1)
                        ldi             ZH, HIGH(@0<<1)
.endmacro

; Store from dataspace to dataspace
; @0 first dataspace
; @1 second dataspace
.macro  STSS
                        lds r24, @1
                        sts @0, r24
.endmacro
```

Appendix D: ui.asm

```
; void print_menu();
; print the initial menu prompt
print_menu:
        LSW             opt_prompt
        rcall   print_string
        LSW             opt_lock
        rcall   print_string
        LSW             opt_unlock
        rcall   print_string
        LSW             opt_reset
        rcall   print_string
        ldi             input_mode, 0x0
        ret

; void print_string(char* str);
; The Z register is expected to contain the low
; and high bytes of the address to a null-terminated
; string in memory.
;
; characters are inserted into serial_out until
; the null character is reached, at which point the
; function returns.
print_string:
        lpm             serial_out, Z+ ; read a character, post-increment Z
        tst             serial_out ; check if it is the null terminator
        breq    print_done ; if we hit the null terminator, jump to the end
        rcall   uart_tx ; print the character
        rjmp    print_string ; Z is already incremented, test/print next char
        print_done:
        ret

; void print_code
; prints current code
print_code:
        lds     serial_out, pwd_buff
        rcall   uart_tx
        lds     serial_out, pwd_buff +1
        rcall   uart_tx
        lds     serial_out, pwd_buff +2
        rcall   uart_tx
        lds     serial_out, pwd_buff +3
        rcall   uart_tx
        ret

; void unlock_prompt()
; prints unlock prompt to screen
lock_prompt:
        LSW             pwd_prompt
        rcall   print_string
        ret

badpw_prmpt:
        LSW             pwd_mismat
        rcall   print_string
        ret

newpw_prompt:
        LSW             pwd_new
        rcall   print_string
        ret

respw_prompt:
        LSW             pwd_msg
        rcall   print_string
        LSW             rdbk_msg
        rcall   print_String
        ret

unlock_prompt:
        LSW             pwd_match
        rcall   print_string
        LSW             unlock_msg
        rcall   print_string
        ret

; void set_reset();
; handles user input to reset the password.
set_reset:
        ldi             light_config, 0x01
        rcall lock_prompt
        rcall   set_lights
        .def pwd_buff_tmp = r22
        .def curr_pwd_tmp = r23

        rcall load_pw_buffer

        lds     pwd_buff_tmp, pwd_buff
        lds     curr_pwd_tmp, curr_pwd
        cp              pwd_buff_tmp, curr_pwd_tmp
```

```
        brne     bad_pwd_res
        lds      pwd_buff_tmp, pwd_buff + 1
        lds      curr_pwd_tmp, curr_pwd + 1
        cp               pwd_buff_tmp, curr_pwd_tmp
        brne     bad_pwd_res
        lds      pwd_buff_tmp, pwd_buff + 2
        lds      curr_pwd_tmp, curr_pwd + 2
        cp               pwd_buff_tmp, curr_pwd_tmp
        brne     bad_pwd_res
        lds      pwd_buff_tmp, pwd_buff + 3
        lds      curr_pwd_tmp, curr_pwd + 3
        cp               pwd_buff_tmp, curr_pwd_tmp
        brne     bad_pwd_res

        rjmp  good_pwd_res

        bad_pwd_res:
                rcall    badpw_prmpt
                ldi              light_config, 0x02
                rcall    set_lights
                LSW              lock_msg
                rcall    print_string
                rjmp     end_pass_res

        good_pwd_res:
                rcall    newpw_prompt
                rcall    load_new_password
                rcall    respw_prompt
                rcall    print_code
                ldi              light_config, 0x04
                rcall    set_lights
                LSW              unlock_msg
                rcall    print_string

        end_pass_res:
                rcall    print_menu
                .undef   pwd_buff_tmp
                .undef   curr_pwd_tmp
        ret

; void set_unlocked();
; handles user input to determine if the
; system should be unlocked.
set_unlocked:
.def pwd_buff_tmp = r22
.def curr_pwd_tmp = r23

        rcall    lock_prompt
        rcall    load_pw_buffer

        lds      pwd_buff_tmp, pwd_buff
        lds      curr_pwd_tmp, curr_pwd
        cp               pwd_buff_tmp, curr_pwd_tmp
        brne     bad_pwd
        lds      pwd_buff_tmp, pwd_buff + 1
        lds      curr_pwd_tmp, curr_pwd + 1
        cp               pwd_buff_tmp, curr_pwd_tmp
        brne     bad_pwd
        lds      pwd_buff_tmp, pwd_buff + 2
        lds      curr_pwd_tmp, curr_pwd + 2
        cp               pwd_buff_tmp, curr_pwd_tmp
        brne     bad_pwd
        lds      pwd_buff_tmp, pwd_buff + 3
        lds      curr_pwd_tmp, curr_pwd + 3
        cp               pwd_buff_tmp, curr_pwd_tmp
        brne     bad_pwd

        rjmp  good_pwd

        bad_pwd:
                rcall    badpw_prmpt
                rjmp     set_unlocked_end

        good_pwd:
                rcall    unlock_prompt
                ldi              light_config, 0x04
                rcall    set_lights
        set_unlocked_end:
        rcall    print_menu
        .undef   pwd_buff_tmp
        .undef   curr_pwd_tmp
        ret

; void test_input
; test input and branch to appropriate menu
test_input:
        cpi              input_mode, 'L'
        breq     is_l
        cpi              input_mode, 'l'
        brne     skip1
        is_l:
        rcall    set_locked
```

```
        skip1:
        cpi             input_mode, 'U'
        breq    is_u
        cpi             input_mode, 'u'
        brne    skip2
        is_u:
        rcall   set_unlocked
        skip2:
        cpi             input_mode, 'R'
        breq    is_r
        cpi             input_mode, 'r'
        brne    skip3
        is_r:
        rcall   set_reset
        skip3:
        ret

; void set_locked();
; lock the system, changing the light to the
; 'locked' light.
set_locked:
        ldi             light_config, 0x02
        LSW             lock_msg
        rcall   print_string
        rcall   set_lights
        rcall   print_menu
        ret

; void clear_lights();
; turns off the LEDs (writes 0x0 to PORTB)
clear_lights:
        ldi             light_config, 0x0
        rcall   set_lights
        ret

; void set_lights();
; turn on LEDs based on the value of light_config
set_lights:
        out             PORTB, light_config
        ret
```

Appendix E: strings.asm

```
;
; strings.asm
;
; null-terminated string constants.
;
;  Created: 3/12/2019 10:16:02 AM
;   Author: Benjamin Garcia, Benjamin LeBrun
;

.equ LF = 0x0A ; newline
.equ CR = 0x0D ; carriage return

; clear string, used to deal with first printed line being garbled
clr_string:     .db     CR,LF,"                        ",CR,LF, 0,0

; menu strings
opt_prompt:     .db CR,LF,"Please enter a command:",CR,LF,0
opt_lock:       .db CR,LF,"L) lock the system",CR,LF,0,0
opt_unlock:     .db     CR,LF,"U) unlock the system",CR,LF,0,0
opt_reset:      .db CR,LF,"R) reset the system's password",CR,LF,0,0

; password strings
pwd_prompt:     .db CR,LF,"Please enter your current password:",CR,LF,0
pwd_mismat:     .db CR,LF,"Error: password did not match.",CR,LF,0,0
pwd_match:      .db CR,LF,"Password matched.",CR,LF,0
pwd_bad:        .db CR,LF,"Error: passwords must consist of four digits in the range 0-9.",CR,LF,0,0
pwd_new:        .db CR,LF,"Please enter your new four digit password:",CR,LF,0,0
pwd_msg:        .db CR,LF,"Password reset.",CR,LF,0
rdbk_msg:       .db CR,LF,"Password changed to:",CR,LF,0,0

; lock strings
lock_msg:       .db CR,LF,"System is now locked.",CR,LF,0

; unlock strings
unlock_msg:     .db CR,LF,"System is now unlocked.",CR,LF,0
```