Objective

The objective of this lab is to demonstrate the use of interrupts to enhance the efficiency and responsiveness of an alarm system on the Dragon-12 Trainer Board, focusing on tasks like display control, keypad scanning, sound generation, and delay management.

Equipment Used:

- Windows PC
- Dragon-12 Trainer

Components Used on Dragon-12 Board:

- 7-segment LED Displays
- KeyPad
- DIP Switch
- LCD Display
- PC Speaker

Introduction

Interrupts are utilized to improve efficiency and responsiveness, replacing polling methods for tasks such as driving the 7-segment displays, scanning the keypad, and generating sound via the PC speaker. Key tasks include updating the Keypad and Segment Display modules and implementing a new Siren Module, with timer interrupts managing delays, display refreshing, key scanning, and waveform generation. This lab demonstrates how interrupts eliminate display flicker and improve system performance, providing practical experience with real-time embedded system design.

Hardware Design

The hardware design involves interfacing the 7-segment displays, LCD, and keypad with the microcontroller ports on the Dragon-12 Trainer. The 7-segment displays will be used to show countdowns and alarm states, while the LCD will display menu options. User inputs from the keypad enable real-time interaction with the alarm system. The design follows the guidelines from the Dragon-12 manual, ensuring correct connections for proper operation of all components as outlined in the lab instructions.

7-Segment Display

This LED-based display can display limited alphanumeric letters or numbers. Because the display update is controlled by interrupts rather than manual function calls, it is connected to the Dragon-12 Trainer board in this lab and shows characters without flickering. To create the desired characters, each section is separately controlled.

The 7-segment display is used to show numbers (0-9) and a few letters, which makes it easy to display information in a clear, simple way. Each segment of the display is controlled by the Dragon-12 board. Resistors are added to protect the segments from too much current.

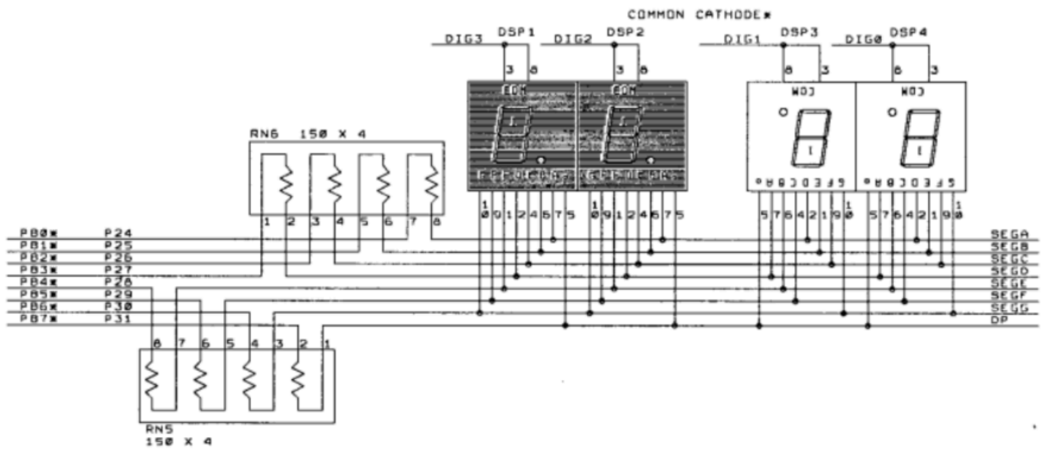
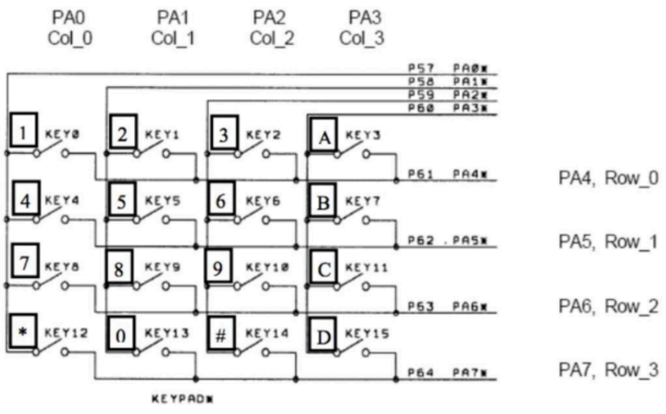


Figure 1: 7-Segment Display

Keypad

A multi-key input device that lets users enter commands or data and is connected to the Dragon-12. The keypad has debounce circuitry to provide reliable key recognition and avoid repeated registrations from a single push, and it regularly scans using timer interrupts.

When a key is pressed, it may produce multiple signals due to "bouncing." Interrupts help manage this by waiting for a stable signal before registering the keypress, ensuring that each press is recorded only once. This process, known as debouncing, prevents errors caused by unintended multiple signals.



Keypad connections:
PA0 connects COL0 of the keypad
PA1 connects COL1 of the keypad
PA2 connects COL2 of the keypad
PA3 connects COL3 of the keypad
PA4 connects ROW0 of the keypad
PA5 connects ROW1 of the keypad
PA6 connects ROW2 of the keypad
PA7 connects ROW3 of the keypad

Figure 2: KeyPad

DIP Switch

A simple input device consisting of multiple toggle switches that act as binary inputs, turning specific pins on or off. These switches allow users to configure or set modes for the alarm system by directly interacting with the microcontroller pins.

PC Speaker

This audio output component generates an alarm sound resembling a siren. Controlled through interrupts and connected to a timer channel on the Dragon-12 board, it produces a waveform whose frequency and duration are managed by the timer when activated.

Overall Electric Circuit Diagram

The figure below shows the overall circuit diagram including the already implemented keypad and 16x2 liquid crystal display, as well as the 7-segment display, dip switches and siren module.

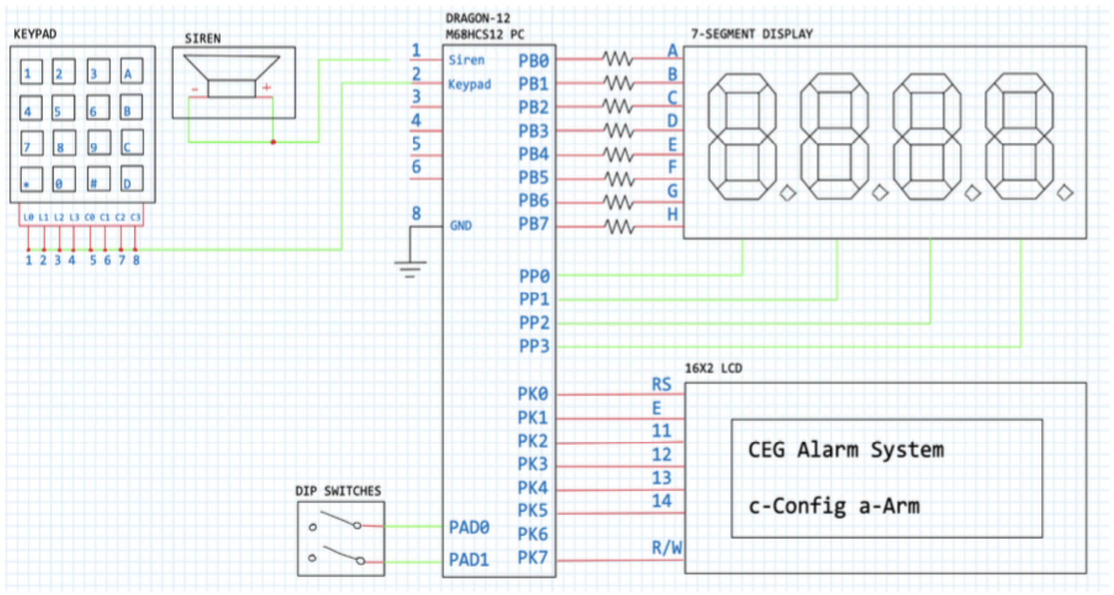


Figure 3: Overall Circuit diagram

Software Design

Module Design and Subroutines:

Keypad Module

The keypad module facilitates communication between the Dragon12 board and its keypad by scanning inputs and converting key presses into ASCII codes. The updated design uses a timer interrupt for keypad scanning, replacing manual debouncing with an Interrupt Service Routine (ISR), which enhances efficiency and response time by scanning only during interrupt intervals.

void initKeyPad(): Configures the keypad input pins and initializes the timer interrupt to enable automatic, periodic scanning.

char readKey(): Waits for the ISR to detect a key press, then translates and returns the corresponding ASCII code.

char pollReadKey(): Performs a non-blocking check for key presses. If a key is detected, it translates and returns the ASCII code.

Segment Display Module

This module manages the 7-segment display, enabling it to show numeric or character values. The update introduces a timer-based interrupt to control the display refresh rate, minimizing flicker and ensuring stable visibility.

void initDisp(): Configures the display ports and initializes the display to its default state.

void clearDisp(): Turns off all segments to clear the display.

void setCharDisplay(): Maps a character (e.g., '0'-'9' or 'A'-'F') to its corresponding 7-segment pattern and updates the specified display unit, allowing seamless updates without affecting other displayed characters.

Siren Module

The siren module controls the PC speaker to emit an alarm sound. It generates a waveform on the speaker using timer interrupts, creating a siren effect that can vary in frequency.

void initSiren(): Sets up the timer for the speaker pin to enable waveform generation.

void turnOnSiren(): Activates the siren by triggering a waveform on the speaker pin through timer interrupts, allowing the alarm to sound continuously without blocking CPU resources.

void turnOffSiren(): Disables the waveform and stops the timer interrupt, silencing the siren completely.

C-Pseudo Code

Keypad Module

- **void initKeyPad()**

The initKeyPad function is responsible for setting up the hardware to detect key presses in an efficient, interrupt-based way. To start, it configures PORTA by designating its lower 4 bits (bits 0-3) as outputs, which represent the rows of the keypad. These outputs are initially set to zero. Pull-up resistors are then enabled on PORTA to stabilize the input signals, ensuring reliable key detection by preventing random fluctuations.

```
/*-------------------------------------------- Function: initKeyPad
Description: Initializes the hardware for the KeyPad Module.
---------------------------------------------*/

void initKeyPad(void)
{
   DDRA = 0xF0;        // Configure the lower 4 bits of port A as output
   PORTA = 0x00;       // Set all the output bits of port A to 0
   PUCR |= 0x01;       // Enable pull-up resistors on port A
   TIOS |= (1 << 4);   // Set output compare mode for timer channel 4
   TIE |= (1 << 4);    // Enable interrupts for timer channel 4
   TC4 = TCNT + TENMSEC; // Set the timer channel 4 timeout for 10
milliseconds
   keyCode = NOKEY;    // Initialize key code to no key pressed
}
```

Pseudo-Code

- **char readKey()**

The readKey function serves to detect key presses and return their ASCII values. It starts by continuously checking the keyCode variable, which is updated through an interrupt set up in initKeyPad when a key is pressed. Once keyCode reflects a key press, the function translates it into an ASCII character using getAscii(keyCode). After conversion, keyCode is reset to NOKEY, allowing readKey to wait for the next key press. Finally, the ASCII character of the pressed key is returned.

```
/*------------------------------------------------ Function: readKey
Description: Waits for a key press and returns its ASCII equivalent.
-------------------------------------------------*/
```

```
char readKey()
{
   char keyChar;
   while(keyCode == NOKEY) { /* wait for key press */ }  // Loop until keyCode
is not NOKEY
   keyChar = getAscii(keyCode);  // Retrieve the ASCII value for the pressed key
   keyCode = NOKEY;  // Reset keyCode after reading the key
   return keyChar;    // Return the ASCII value of the pressed key
}
```

Pseudo-Code

- **char pollReadKey()**

The pollReadKey function is designed to check for a key press and return its ASCII value if a key is detected; if no key is pressed, it simply returns NOKEY. When keyCode is equal to NOKEY, the function recognizes that no key press has occurred and immediately returns NOKEY. If a key has been pressed, however, pollReadKey converts keyCode into its ASCII form using getAscii(keyCode), then resets keyCode back to NOKEY to be ready for the next press, and finally returns the ASCII character.

```
/*------------------------------------------------
Function: pollReadKey
Description: Checks for a key press and returns its ASCII
equivalent if present; otherwise returns NOKEY.
-------------------------------------------------*/
char pollReadKey()
{
   char keyChar;
   if (keyCode == NOKEY) {
      keyChar = NOKEY;  // No key pressed, return NOKEY
   } else {
      keyChar = getAscii(keyCode);  // Get ASCII value of pressed key
      keyCode = NOKEY;  // Reset keyCode after reading the key
   }
   return keyChar;  // Return ASCII value of key or NOKEY
}
```

Pseudo-Code

Segment Display Module

- **void initDisp()**

The initDisp function prepares the hardware to operate the 7-segment displays in the alarm system. It configures Port B and certain bits of Port P as outputs to manage the display data and select which display to activate. Initially, it disables all displays by setting PTP, and then clears any active display segments with a call to clearDisp(). The function also sets up timer channel 1 to generate interrupts at intervals defined by DISP_TIMEOUT, allowing the display to refresh periodically

```
/*-------------------------------------------- Function: initDisp
Description: Initializes the hardware for the 7-segment displays.
---------------------------------------------*/
void initDisp(void) {
   // Configure ports B and P to control the displays
   DDRB = 0xFF;         // Set port B as output
   DDRP |= 0x0F;       // Set the lower 4 bits of port P as output
   PTP |= 0x0F;        // Disable all displays initially
   clearDisp();        // Clear all displays

   // Set up timer channel to generate interrupts
   // Assuming the timer is enabled elsewhere with 1 1/3 microsecond ticks for
controlling displays
   TIOS |= 0b00000010;  // Set output compare for timer channel 1
   TIE |= 0b00000010;   // Enable interrupt for timer channel 1
   TC1 = TCNT + DISP_TIMEOUT;  // Set timeout for timer channel 1
}
```

Pseudo-Code

- **void clearDisp()**

The clearDisp function is responsible for turning off all segments of the 7-segment displays, effectively blanking each display. It does this by setting every element in the codes array, which controls each display's segments, to zero. This action ensures that no segments are lit, keeping the displays blank.

```
/*-------------------------------------------- Function: clearDisp
Description: Clears all the displays. ---------------------------------------------*/
void clearDisp(void)
{
   int index;
   for (index = 0; index < NUMDISPS; index++) {
      codes[index] = 0;  // Reset each display code to 0
   }
}
```

Pseudo-Code

- **void setCharDisplay()**

The function setCharDisplay converts an ASCII character ch into the corresponding segment code for a 7-segment display, which it then stores in the codes array at the index specified by dispNum. The getCode(ch) function provides the segment code for the given character. To maintain the existing decimal point on the display, the expression codes[dispNum] & 0x80 ensures that the decimal bit is preserved while updating the rest of the code with the new character's segment pattern. In an interrupt-driven alarm system, this function is crucial for dynamically refreshing the displays with updated data (such as numbers or status indicators) while retaining the decimal point when necessary.

```
/*-------------------------------------------- Function: setCharDisplay
Description: Receives an ASCII character (ch) and translates
it to the corresponding code to display on a 7-segment display.
The code is stored in the appropriate element of the 'codes' array for the identified
display (dispNum).
----------------------------------------------*/
void setCharDisplay(char ch, byte dispNum) {
   byte displayCode;
   displayCode = getCode(ch);  // Get the 7-segment code for the character
   codes[dispNum] = displayCode | (codes[dispNum] & 0x80);  // Update the
display code with the mask
}
```

Pseudo-Code

Siren Module

- **void initSiren()**

The initSiren function sets up Timer Channel 5 (TC5) to work in output-compare mode. This mode allows TC5 to control an output signal based on specific time intervals. By configuring TC5 in this way, the function enables the alarm system to create a siren sound by turning an output on and off at regular intervals, forming the required waveform for the siren. This setup is important for an interrupt-driven alarm system, as it allows the siren to trigger automatically without needing constant attention from the main program, making the system more efficient for real-time alerts.

```
void initSiren() {
TIOS |= 0b00100000; // Set TC5 to output-compare }
```

Pseudo-Code

- **void turnOnSiren()**

The turnOnSiren function starts the alarm by setting Timer Channel 5 (TC5) to output a high pulse, initiating the siren sound. It then configures TC5 to toggle between high and low at set intervals, based on the HIGH_MS delay constant. Enabling interrupts allows the system to manage the toggles automatically, keeping the siren on and off without manual control. This setup is key for an efficient, responsive alarm in an interrupt-driven system.

```
#define HIGH 1
#define LOW 0

int levelTC5;  // Current level of TC5 output pin

void turnOnSiren() {
   TCTL1 |= 0x0C;      // Set TC5 output to high on output-compare event
   CFORC = 0x20;       // Force an event to set TC5 pin high immediately
   levelTC5 = HIGH;    // Update TC5 level to HIGH
   TCTL1 &= 0xF7;      // Reset to toggle mode for TC5
   TC5 = TCNT + HIGH_MS;  // Set TC5 timeout interval based on HIGH_MS
   TIE |= 0x20;        // Enable interrupt for TC5
}
```

Pseudo-Code

- **void turnOffSiren()**

The turnOffSiren function stops the alarm by setting Timer Channel 5 (TC5) output to low, silencing the siren. It first disables interrupts on TC5 to prevent further toggling, then forces the TC5 pin to output low. This function is essential for quickly and efficiently deactivating the siren in an interrupt-driven alarm system, maintaining control over the alert.

```
void turnOffSiren() {
   TIE &= 0xDF;       // Disable interrupt for TC5
   TCTL1 |= 0x08;     // Set to output low on pin 5 at output-compare event
   TCTL1 &= 0xFB;     // Clear the output toggle bit to ensure low output
   CFORC = 0x20;      // Force an event to set TC5 pin low immediately
}
```

Pseudo-Code

Code Testing for Required Modules

Alarm System Module

The Alarm System Module manages the operation of an alarm system, controlling sensors, displays, and audio signals. It triggers alarms, activates visual or auditory indicators, and monitors system status to alert users of potential security breaches.

- **delay.c**

```
/*------------------------------------------------------
File: Delay.c
Description: Delay module using Timer Channel 0
------------------------------------------------------*/
#include "mc9s12dg256.h"
#include <stddef.h>
#include "Delay.h"

// Constants for delay
#define TENTH_MS 75 // Timer value for 0.1ms delay (75*1 1/3 micro-sec)

// Global Variables
static volatile int delayCounter; // Global counter for blocking delay
static volatile int *externalCounter = NULL; // Pointer to external counter
static volatile int timerCounter;

/*---------------------------------------------------
Function: initDelay
Description: Initializes Timer Channel 0 for 0.1ms increments
---------------------------------------------------*/
void initDelay(void) {
   TIOS_IOS0 = 1;       // Set TC0 to output-compare mode
   TIE_C0I = 0x01;      // Enable interrupt on TC0
   TC0 = TCNT + TENTH_MS; // Set TC0 to generate a 0.1ms delay
   timerCounter = 10;   // Initialize the interrupt counter to create 1ms delay
}

/*---------------------------------------------------
Function: setCounter
Description: Sets the address of an external counter.
Pass NULL to disable the counter.
---------------------------------------------------*/
void setCounter(volatile int *extCounterPtr) {
   externalCounter = extCounterPtr; // Assign the pointer to the external counter
}

/*---------------------------------------------------
Function: delayms
Description: Delays for a specified number of milliseconds (blocking delay)
```

```
-----------------------------------------------------*/
void delayms(int num) {
   delayCounter = num;  // Set the delay counter to the desired delay time
   while (delayCounter) { /* wait */ } // Loop until the counter reaches zero
}

/*--------------------------------------------------
Interrupt: tc0_isr
Description: Interrupt service routine that decrements the
delayCounter and external counter (if set) every 1ms.
-----------------------------------------------------*/
void interrupt VectorNumber_Vtimch0 tc0_isr(void) {
   timerCounter--;  // Decrease the ISR counter
   if (timerCounter == 0) { // Check if the counter has reached zero
      timerCounter = 10;  // Reset the ISR counter to 10 for 1ms interval
      delayCounter--;     // Decrease the main delay counter
      if (externalCounter != NULL) {
         (*externalCounter)--; // Decrease external counter if not NULL
      }
   }
   TC0 = TC0 + TENTH_MS;  // Reset the timer interrupt by updating TC0
}
```

C-code

- **keyPad.c**

```
/*-------------------------------------------
Function: initKeyPad
Description: Initializes hardware for the KeyPad Module.
----------------------------------------------*/
void initKeyPad(void) {
   DDRA = 0xF0;  // Set lower 4 bits of port A to output
   PORTA = 0x00;  // Set all output bits of port A to 0
   PUCR |= 0x01;   // Enable pullup resistors on port A
   TIOS |= BIT4;   // Set output compare for timer channel 4
   TIE |= BIT4;    // Enable interrupt on timer channel 4
   TC4 = TCNT + TENMSEC;  // Set timeout for timer channel 4
   keyCode = NOKEY; // Initialize key code to no key
}

/*----------------------------------------------
Function: readKey
Description: Waits for a key and returns its ASCII equivalent.
-------------------------------------------------*/
char readKey() {
   char ch;
   while (keyCode == NOKEY);  // Wait until key is pressed
   ch = getAscii(keyCode);   // Retrieve the ASCII value of the key
   keyCode = NOKEY;          // Reset keyCode after reading key
   return ch;                // Return the ASCII of the key pressed
}

/*-----------------------------------------------
Function: pollReadKey
Description: Checks for a key, returns its ASCII equivalent if pressed; otherwise
returns NOKEY.
-----------------------------------------------*/
char pollReadKey() {
   char ch;
   if (keyCode == NOKEY) {
      ch = NOKEY;  // Return NOKEY if no key is pressed
   } else {
      ch = getAscii(keyCode);  // Get ASCII of the key
      keyCode = NOKEY;         // Reset keyCode after reading key
   }
   return ch;  // Return key ASCII or NOKEY
}

/*-----------------------------------------------
Function: key_isr
Description: Display interrupt service routine that checks keypad every 10 ms.
-----------------------------------------------*/
void interrupt VectorNumber_Vtimch4 key_isr(void) {
   static byte state = WAITING_FOR_KEY;  // Initial state: waiting for key
   static byte code;
   switch (state) {
      case WAITING_FOR_KEY:
         code = PORTA;  // Get value from PORTA
         if (code != 0x0F) state = DEB_KEYPRESS;  // If key is pressed, debounce
         break;
      case DEB_KEYPRESS:
         if (PORTA != code) {
            state = WAITING_FOR_KEY;  // If PORTA changes, go back to waiting
         } else {
            code = getKCode();  // Get key code
            state = WAITING_FOR_REL;  // Wait for key release
         }
         break;
      case WAITING_FOR_REL:
         if (PORTA == 0x0F) state = DEB_REL;  // If key released, debounce
release
         break;
      case DEB_REL:
         if (PORTA != 0x0F) {
            state = WAITING_FOR_REL;  // If key still held, wait for release
         } else {
            keyCode = code;  // Set detected key code
            state = WAITING_FOR_KEY;  // Go back to waiting for next key
         }
         break;
   }
   TC4 = TC4 + TENMSEC;  // Reset timer for next interrupt
}

/*------------------------------------------------
Function: getKCode
Description: Gets the key code corresponding to a keypress from PORTA.
------------------------------------------------*/
byte getKCode() {
   volatile byte code;
   PORTA = ROW1;  // Set PORTA to row 1
   if (PORTA == ROW1) {
      PORTA = ROW2;  // Set PORTA to row 2
      if (PORTA == ROW2) {
         PORTA = ROW3;  // Set PORTA to row 3
         if (PORTA == ROW3) {
            PORTA = ROW4;  // Set PORTA to row 4
         }
      }
   }
   code = PORTA;  // Get the key code from PORTA
   PORTA = 0x00;  // Set all output pins to low
   return code;   // Return the detected key code
}

/*------------------------------------------------
Function: getAscii
Description: Converts the key code from PORTA to its corresponding ASCII value.
------------------------------------------------*/
char getAscii(byte cd) {
   int i;
   char ch = BADCODE;  // Default to bad code
   for (i = 0; kCodes[i] != BADCODE; i++) {  // Loop through key codes
      if (kCodes[i] == cd) {  // If key code matches
         ch = aCodes[i];   // Get corresponding ASCII
         break;  // Exit loop
      }
   }
   return ch;  // Return ASCII or bad code }
```

C code

LCD Display Module

The LCD Display Module manages communication with the LCD screen using both assembly and C code. The assembly part handles low-level tasks like initialization, clearing the screen, and displaying characters in 4-bit mode. The C code provides higher-level functions for printing and positioning text, as well as padding strings to fit the display.

This module works with the alarm system and other components by displaying important information such as countdowns, alarm status, and error messages. It updates the display by receiving data through function calls like printLCDStr and putLCDChar. The screen is refreshed either after timed delays or in response to system interrupts, making it an essential part of the user interface for real-time feedback and interaction.

- **lcd.asm**

```
;***********************************************************
;* File: lcd.asm
;* Assembly language routines for C function calls
;* for manipulating the LCD
;***********************************************************
; External symbols referenced XREF delayms
; Internal symbols defined for access
XDEF data8, lcd_init, clear_lcd, set_lcd_addr
XDEF type_lcd

; Include derivative specific macros
INCLUDE 'mc9s12dg256.inc'

; Code section
.text

; Initialize LCD
lcd_init:
   ldx #init_codes        ; Point to initialization codes
   ldaa #$FF              ; Set all bits of DDRK to output (PORTK)
   staa DDRK
   pshb                   ; Save register B to stack
   ldab 1,x+              ; Load first initialization byte from init_codes
   jsr write_instr_byte   ; Write instruction byte to LCD
   pshd
   pshx
   ldd #5                 ; Repeat initialization 5 times
lcdi1:
   ldaa 1,x+              ; Load next initialization byte
   jsr write_instr_byte   ; Write instruction byte
   pulx                   ; Restore X
```

```
    puld
    decb                    ; Decrement counter
    bne lcdi1               ; Loop if counter is not zero
    pulb                    ; Restore B
    rts

; Write instruction byte to LCD
instr8:
    tba                     ; Transfer A to B
    jsr sel_inst            ; Select instruction mode
    jsr write_instr_byte    ; Write instruction byte to LCD
    ldd #10                 ; Delay for 10ms
    jsr delayms
    rts

; Write data byte to LCD
data8:
    tba                     ; Transfer A to B
    jsr sel_data            ; Select data mode
    jsr write_data_byte     ; Write data byte
    ldd #10                 ; Delay for 10ms
    jsr delayms
    rts

; Set LCD address
set_lcd_addr:
    orab #$80               ; Set address to B
    jsr write_instr_byte    ; Write address to LCD
    rts

; Clear LCD
clear_lcd:
    ldaa #$01               ; Load clear command
    jsr write_instr_byte    ; Send clear command
    jsr delayms
    jsr write_instr_byte    ; Send another instruction byte
    jsr delayms
    rts

; Display an ASCII string on LCD
type_lcd:
    pshx                    ; Save X register (pointer to string)
next_char:
    ldaa 1,x+               ; Load next character from string
    beq done                ; If null (end of string), quit
    jsr write_data_byte     ; Display the character
    pshx                    ; Save X before delay
    ldd #10                 ; Delay for 10ms
    jsr delayms
    pulx                 ; Restore X
    bra next_char          ; Loop to next character
done:
    pulx                 ; Restore X
    rts

; Write instruction upper nibble to LCD
write_instr_nibble:
    anda #$F0            ; Mask to get upper nibble
    lsra                 ; Shift right by 2 to position in pk2-pk5
    oraa #$02            ; Set E=1 and RS=0 (instruction mode)
    staa PORTK             ; Write to PORTK
    ldy #10              ; Wait for write to complete
win:
    dey
    bne win
    anda #$FC            ; Set E=0 (end of instruction)
    staa PORTK             ; Write to PORTK
    rts

; Write data upper nibble to LCD
write_data_nibble:
    anda #$F0            ; Mask to get upper nibble
    lsra                 ; Shift right by 2 to position in pk2-pk5
    oraa #$03            ; Set E=1 and RS=1 (data mode)
    staa PORTK             ; Write to PORTK
    ldy #10              ; Wait for write to complete
wdn:
    dey
    bne wdn
    anda #$FD            ; Set E=0 (end of data)
    staa PORTK             ; Write to PORTK
    rts

; Write instruction byte to LCD
write_instr_byte:
    psha                 ; Push A to stack
    jsr write_instr_nibble  ; Write upper nibble
    pula                 ; Pop A from stack
    asla                 ; Shift left by 4 (to get lower nibble)
    asla
    asla
    asla
    jsr write_instr_nibble  ; Write lower nibble
    rts

; Write data byte to LCD
write_data_byte:
    psha                 ; Push A to stack
    jsr write_data_nibble   ; Write upper nibble
    pula                    ; Pop A from stack
    asla                   ; Shift left by 4 (to get lower nibble)
    asla
    asla
    asla
    jsr write_data_nibble   ; Write lower nibble
    rts

.rodata
; Initialization codes for 4-bit mode
; Uses only data in high nibble
init_codes:
    fcb 12                 ; Delay for reset
    fcb $30                 ; First reset code (must delay 4.1ms)
    fcb $30                 ; Second reset code
    fcb $20                 ; Third reset code (4-bit mode)
    fcb $20                 ; Set 2 lines, 5x7 dot format
    fcb $80                 ; Cursor increment, disable display shift
    fcb $00                 ; Display on, cursor off, no blinking
    fcb $60                 ; Clear display memory, set cursor to home
    fcb $00                 ; Wait 100us after second reset
    fcb $C0                  ; Wait 40us after sending each nibble
    fcb $00
    fcb $10                 ; Final reset code (4-bit mode)
```

Assembly Code

- **lcdDisp.c**

```
/*-------------------------------------
File: lcdDisp.c (LCD Display Module)
Description: C Module that provides display functions on the
LCD. It makes use of the LCD ASM
Module developed in assembler.
-------------------------------------*/
#include <mc9s12dg256.h>   // Microcontroller header for mc9s12dg256
#include "lcd_asm.h"       // Includes the LCD assembly header

// Some Definitions
#define NUM_LINES 2        // Defines the number of lines on the LCD
#define LINE_OFFSET 40     // Defines the offset for the second line
#define LINE_SIZE 16       // Defines the size of each line (16 characters)

// Prototypes of local functions
void padLCDString(char *str, char *newstr, byte size);  // Function prototype to pad
string for LCD display
```

```
/*--------------------------
Function: initLCD
Parameters: None.
Returns: nothing
Description: Initializes the LCD hardware by
calling the assembler subroutine.
---------------------------*/
void initLCD(void) {
   lcd_init(); // Calls the assembly function to initialize the LCD
}

/*--------------------------
Function: printLCDStr
Parameters:
   - str: Pointer to string to be printed (only 16 chars are printed)
   - lineno: 0 for first line, 1 for second line
Returns: nothing
Description: Prints a string on the LCD display on one of the two lines.
The string is padded with spaces to erase any existing characters.
---------------------------*/
void printLCDStr(char *str, byte lineno) {
   char newstr[LINE_SIZE + 1]; // Creates a new string buffer with space for the
null terminator

   if (lineno < NUM_LINES) { // Only 2 lines in display, checks if lineno is valid
      set_lcd_addr(lineno * LINE_OFFSET); // Line 1 starts at 40, sets LCD address
to start of the line
      padLCDString(str, newstr, LINE_SIZE); // Pads string to fit within LCD line
size
      type_lcd(newstr); // Displays padded string on LCD
   }
   // No error is generated if lineno is invalid (out of range)
}

/*--------------------------
Function: padLCDString
Parameters:
   - str: Original string to pad
   - newstr: New string for padding
   - size: Size of the new string
Returns: nothing
Description: Copies the string referenced by str to the buffer newstr and
pads with spaces to fill the buffer.
---------------------------*/
void padLCDString(char *str, char *newstr, byte size) {
   int i = 0; // Index to iterate through the string

   // Copy characters from str to newstr
   while (i < size) {
        if (*str == '\0') break; // Stops if the original string ends
        *newstr++ = *str++;      // Copy character from str to newstr
        i++;                     // Increment index
    }

    // Pad remaining space with spaces if newstr is shorter than size
    while (i < size) {
        *newstr++ = ' ';         // Add space to newstr
        i++;                     // Increment index
    }

    *newstr = '\0';              // Terminate the string with a null character
}

/*--------------------------
Function: putLCDChar
Parameters:
    - ch: Character to be printed
    - lineno: 0 for first line, 1 for second line
    - chpos: Position from 0 to 15 (character position on the line)
Returns: nothing
Description: Prints the character at the specified position on the line.
---------------------------*/
void putLCDChar(char ch, byte lineno, byte chpos) {
    byte adr; // Address to store position

    // Check if line and position are valid
    if (lineno < NUM_LINES && chpos < LINE_SIZE) {
        adr = lineno * LINE_OFFSET + chpos; // Calculate the address based on line
number and character position
        set_lcd_addr(adr);               // Set the LCD address to the calculated position
        data8(ch);                       // Send the character to the LCD
    }
}
```

C-code

Segment Display Module

This module controls a 4-digit 7-segment display by updating each segment every 50 milliseconds using a timer interrupt. It converts ASCII characters to display codes, handles the decimal point, and cycles through the digits to display the desired output.

Integrated with the alarm system, the 7-segment display shows key status information during phases like arming or disarming. It displays countdowns, mode (arming/disarming), and error messages, all refreshed by the timer interrupt. The setCharDisplay function enables the display of characters like "A" for arming or "D" for disarming, while turnOnDP and turnOffDP manage the decimal point for additional indicators, such as active status.

- **intDisp.c**

```
/*--------------------------------------------
Function: initDisp
Description: Initializes hardware for the
7-segment displays.
----------------------------------------------*/
void initDisp(void) {
   // Set up port B and P to control displays
   DDRB = 0xFF;        // Set output direction for port B (all pins as output)
   DDRP |= 0x0F;       // Set output direction for port P bits 0 to 3 (control pins for
displays)
   PTP |= 0x0F;        // Disable all displays by setting corresponding pins high (turn
off)

   clearDisp();        // Clears all displays

   // Set up timer channel to generate interrupts
   // Assume timer is enabled elsewhere with 1 1/3 microsec ticks for controlling
displays
   TIOS |= 0b00000010;  // Set output compare for TC1
   TIE |= 0b00000010;   // Enable interrupt on TC1
   TC1 = TCNT + DISP_TIMEOUT;  // Enable timeout on channel 1 (sets up the
next interrupt)
}

/*--------------------------------------------
Function: clearDisp
Description: Clears all displays.
----------------------------------------------*/
void clearDisp(void) {
   int i;
   // Reset all display codes to 0 (blank displays)
   for (i = 0; i < NUMDISPS; i++) {
      codes[i] = 0;
   }
}

/*--------------------------------------------
Function: setCharDisplay
Description: Receives an ASCII character (ch)
and translates it to the corresponding code for
7-segment display. The code is stored in the
appropriate element of the codes array for the
identified display (dispNum).
----------------------------------------------*/
void setCharDisplay(char ch, byte dispNum) {
   byte code;
   code = getCode(ch);  // Get the corresponding 7-segment display code for the
character
   codes[dispNum] = code | (codes[dispNum] & 0x80);  // Preserve the decimal
point (if any)
}

/*----------------------------------------------
Function: getCode
Description: Translates an ASCII character (ch)
to a 7-segment display code. Returns 0 (blank)
if the character is not in the table.
-----------------------------------------------*/
byte getCode(byte ch) {
   byte code = 0;
   byte i;

   // Search for the character in the display table
   for (i = 0; i < NUMCHS && code == 0; i++) {
      if (ch == dispTbl[i].ascii) {
         code = dispTbl[i].code;  // Found the corresponding display code
      }
   }

   return code;  // Return the display code or 0 if not found
}

/*----------------------------------------------
Function: turnOnDP
Description: Turns on the decimal point of the
2nd display from the left (dNum).
-----------------------------------------------*/
void turnOnDP(int dNum) {
   codes[dNum] = codes[dNum] | 0x80;  // Sets bit 7 to 1 to turn on the decimal
point
}

/*----------------------------------------------
Function: turnOffDP
Description: Turns off the decimal point of the
2nd display from the left (dNum).
-----------------------------------------------*/
void turnOffDP(int dNum) {
   codes[dNum] = codes[dNum] & 0x7F;  // Clears bit 7 to 0 to turn off the decimal
point
}

/*------------------------------------------------
Interrupt: disp_isr
Description: Display interrupt service routine that
updates the displays every 50 ms.
```

```
-------------------------------------------------*/
void interrupt VectorNumber_Vtimch1 disp_isr(void) {
   static byte dNum = 0;  // Keeps track of the current display number to refresh

   PORTB = codes[dNum];  // Update the corresponding 7-segment display
(PORTB controls the segments)

   byte enable = PTP;     // Read the current state of port P
   enable &= 0xF0;        // Mask out the lower four bits (to preserve control pins for
displays)
   PTP = enable | enableCodes[dNum];  // Set the lower four bits to the appropriate
display enable code

   dNum = (dNum + 1) % NUMDISPS;  // Move to the next display, wrapping
around if necessary

   // Set up the next interrupt to trigger in DISP_TIMEOUT ticks (also clears the
interrupt)
   TC1 = TC1 + DISP_TIMEOUT;
}
```

C-code

Siren Module

This code controls a siren using timer interrupts. The sirenISR interrupt service routine (ISR) toggles the siren's state between high and low, alternating the signal to produce sound. The levelTC5 variable tracks the current state, and the timer is set to trigger the ISR with a "high" or "low" duration (HIGH_MS and LOW_MS).

The main program initializes the siren with initSiren and activates it with turnOnSiren. Once the siren is on, the ISR handles the timing and toggling of the signal, allowing the main program to focus on other tasks without needing to manage the siren directly.

- **siren.c**

```
/*-----------------------------------------------
File: siren.c
Description: The Siren module.
-----------------------------------------------*/
#include "mc9s12dg256.h"

// Definitions for high and low signal durations (in timer ticks)
#define HIGH_MS 300    // 300 * 1 1/3 micro-sec = 0.400 ms
#define LOW_MS 600     // 600 * 1 1/3 micro-sec = 0.800 ms

// Prototypes of local functions
```

```
void interrupt VectorNumber_Vtimch5 sirenISR(void);

// Setup TC5 for output compare mode, needed for controlling the siren
void initSiren(void) {
   TIOS |= 0b00100000;  // Set TC5 to output-compare mode so we can control
output
}

// Defines high and low levels for TC5 signal
#define HIGH 1
#define LOW 0

int levelTC5;  // Stores the current level of TC5 signal (high or low)

// Turns on the siren by generating a pulse at a high level
void turnOnSiren(void) {
   TCTL1 |= 0b00001100;   // Set TC5 pin high when an output-compare event
happens
   CFORC = 0b00100000;    // Force TC5 to immediately generate an event (pin
goes high)

   levelTC5 = HIGH;       // Record that TC5 is high now
   TCTL1 &= 0b11110111;   // Change pin behavior to toggle on subsequent
interrupts
   TC5 = TCNT + HIGH_MS;  // Set next event to occur after the high duration
   TIE |= 0b00100000;     // Enable interrupt for TC5 to keep toggling
}

// Stops the siren by turning the signal to low and disabling interrupt
void turnOffSiren(void) {
   TIE &= 0b11011111;     // Disable interrupt for TC5 to stop toggling
   TCTL1 |= 0b00001000;   // Set TC5 pin to low when an output-compare event
happens
   TCTL1 &= 0b11111011;   // Change pin behavior to set low instead of toggling
   CFORC = 0b00100000;    // Force TC5 to immediately generate an event (pin
goes low)
}

// Interrupt service routine that manages the toggling of the siren sound
void interrupt VectorNumber_Vtimch5 sirenISR(void) {
   if (levelTC5 == HIGH) {  // If the current level is high
      TC5 += LOW_MS;       // Set next event after low duration
      levelTC5 = LOW;      // Change the level to low
   } else {  // If the current level is low
      TC5 += HIGH_MS;      // Set next event after high duration
      levelTC5 = HIGH;     // Change the level to high
   }
}
```

C-code

Snapshots of the Alarm System in Action

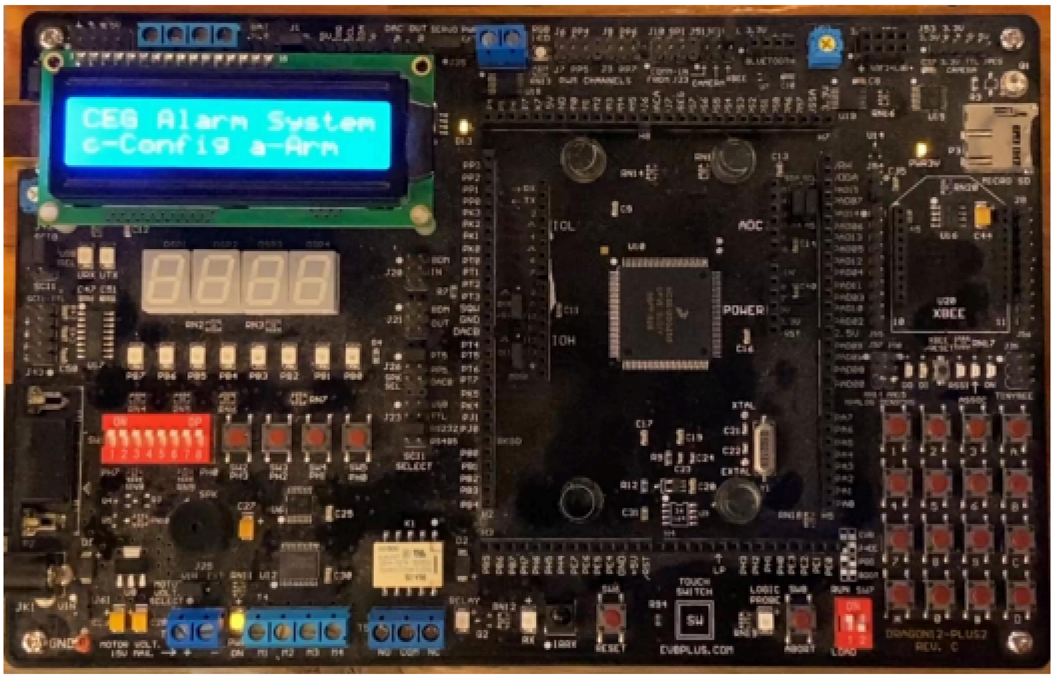


Figure 4 - Alarm System Snapshot: Main

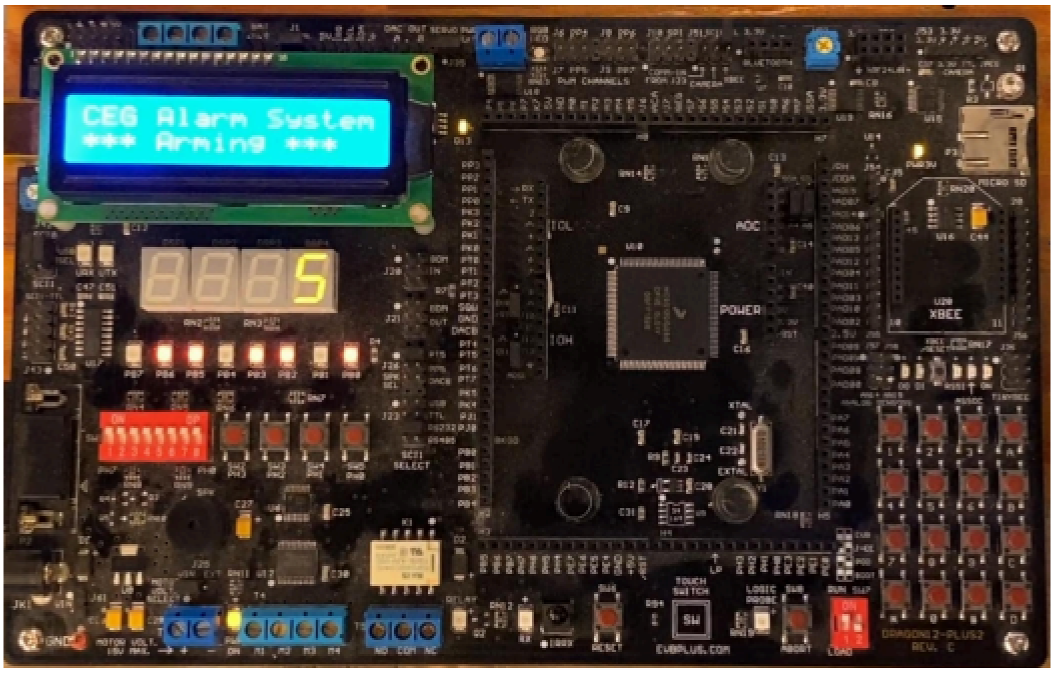


Figure 5 - Alarm System Snapshot: Arming

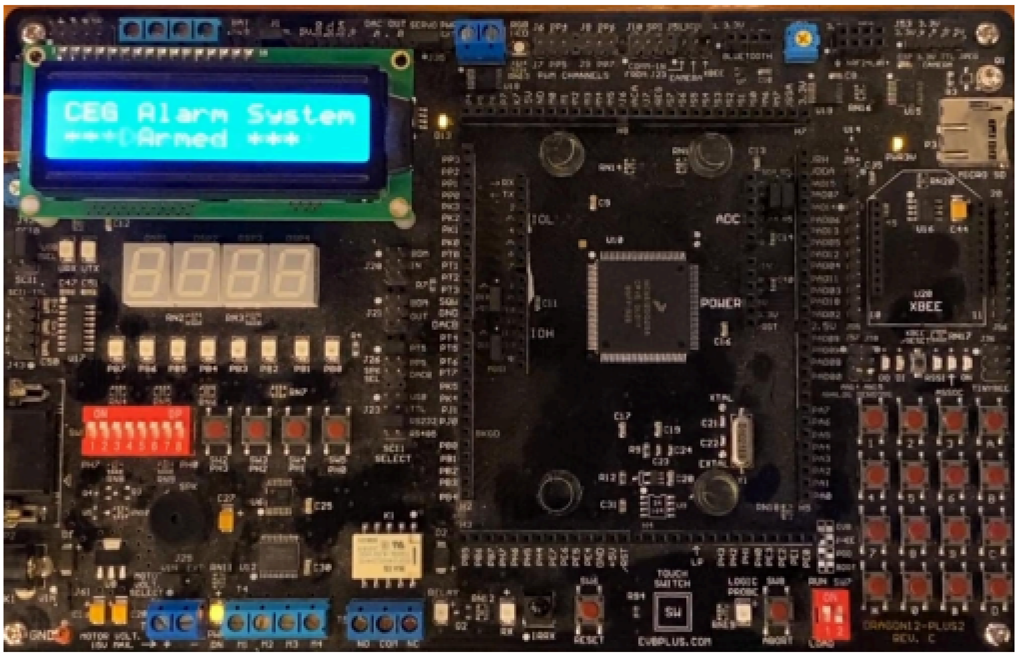


Figure 6 - Alarm System Snapshot: Armed

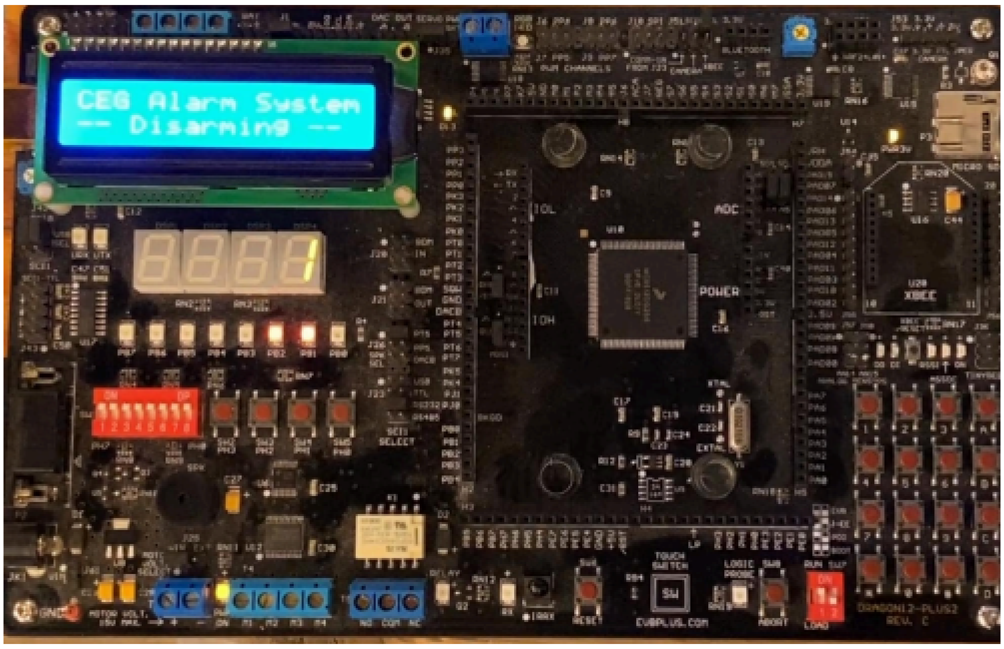


Figure 7 - Alarm System Snapshot: Disarming