CEG 3136 – Computer Architecture II
Alarm System Simulation Design Document

Description and Problem Specification

This document presents the design of an alarm system simulation that is implemented in assembler and executed on a Dragon 12 Plus Trainer. The design is based on modular and structure programming.

The alarm system has been designed with the following features:

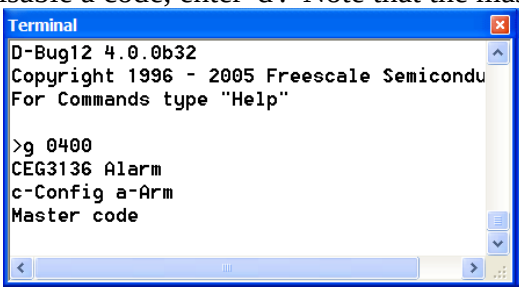
- Configuration of a master code (initially the master code is set to 0000). This master code can (and should) be changed. Codes are 4 digits long, with values from 0000 to 9999.
- Addition of secondary codes. Up to 4 additional codes can be added, for a total of 5 codes.
- Arming the alarm system: entering any of the codes will set the alarm. There should be a delay of 15 seconds before alarm is enabled (to allow opening of a door to leave). This feature shall be completed during lab 1.
- Trigger alarm - when the alarm is set, typing one of the following characters triggers the alarm instantly (bel characters are sent to the terminal): b, c, d, e (typing in a character represents the opening of a door/window). When the alarm has been triggered, it can be turned off by entering an appropriate code.
- Disarming the alarm system: Typing in the character 'a' represents opening the front door, and 15 seconds are allowed to type in one of the alarm codes to disable the alarm system. If a code is not entered within the time frame, the alarm is triggered.

Overall Design

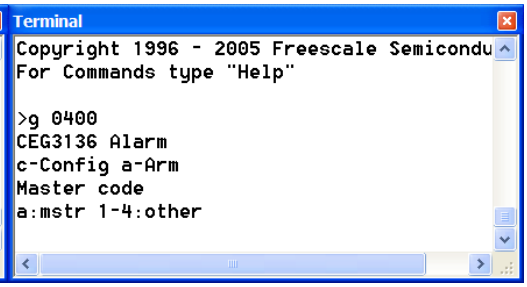
The Dragon 12 Plus Trainer interacts with a terminal (the terminal of the MiniIDE). The following shows some screen shots of how to configure the alarm codes.

The following steps can be used to update a code:

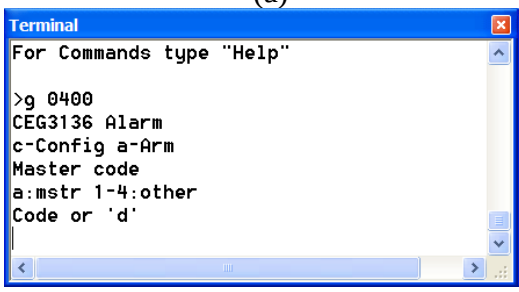
a) To change the master code or add/change a code, type in 'c' (for configure).
b) The master code must be entered (initially the code is 0000).
c) Select the code to update, either 'a' for the master code, or the number 1 to 4, to update the additional codes.
d) To disable a code, enter 'd'. Note that the master code cannot be disabled.



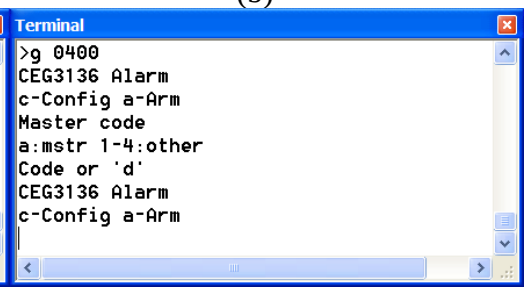
(a)



(b)

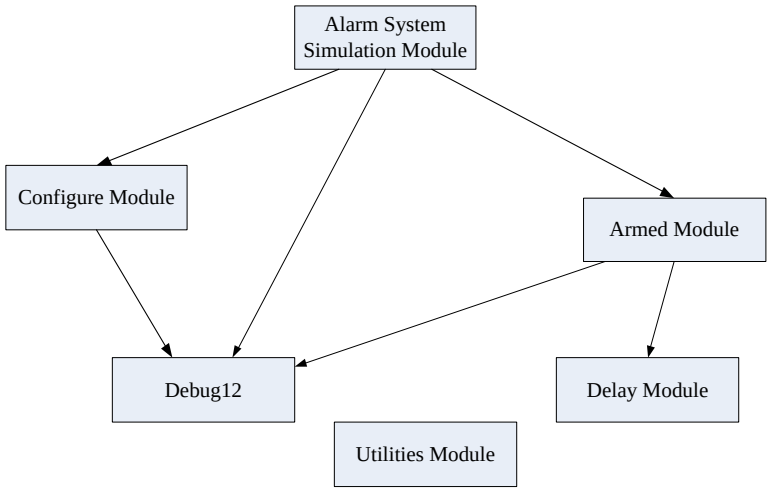


(c)



(d)

Software projects are sub-divided into modules. The algorithm design shall be developed using the C programming language with is then manually translated into assembler. The Alarm Simulation software is divided into five modules as shown in the diagram below:



1. **Alarm System Simulation Module**: This module consists of the main routine that controls the overall software, using the subroutines provided by the other modules. It displays the main menu for the user.
2. **Configure Module**: This module provides a subroutine to configure codes.
3. **Armed Module**: This module provides subroutines to arm and disarm the alarm system. It will monitor the system and trigger the alarm under appropriate conditions. It will also deal with turning off the alarm once it has been triggered.
4. **Delay Module**: This module provides subroutines to create delays in multiples of 1 ms.
5. **Debug12 Module**: This module is provided by firmware existing on the Dragon 12 Trainer. A number of subroutines exist for accessing the terminal, setting breakpoints, and more (see the lab manual for a more complete description). For the Alarm System Simulation project, the subroutines for manipulating the terminal (printing to the display and reading from the keyboard) are of interest.
6. **Utilities Module**: This module contains some general subroutines that can be used by any of the other modules. For example, a subroutine called *isdigit()* exists to check if a character is a digit 0 to 9.

Global Data Structures:

The Alarm System Simulation uses one global data structure, the *code* array, to represent and array of 2 byte integers. The array contains 5 values, where the first value is the master code, and the other 4 values are secondary codes. Each element of the array is 4 bytes long and contains either a value from 0 to 9999 (0 to 9) or the value $FFFF_{16}$ when disabled; note that the master code cannot be disabled.

In C (see alrmSimul.c):

```
// data structure to represent the valid alarm codes.
#define NUMCODES 5
int codes[NUMCODES] = {0,0xffff, 0xffff, 0xffff, 0xffff };
```

In assembler (see alarmSimul.asm):

```
; data structure to represent the valid alarm codes.
codes dc.w 0000,FFFF, FFFF, FFFF, FFFF    ; an array of alarm codes
```

The following strings are used in the program and are defined in the modules where the strings are used:

In C:

```
// Alarm System Simulation Module (alrmSimul.c)
#define MENU "CEG3136 Alarm\nc-Config a-Arm\n"
```

In assembler:

```
// Alarm System Simulation Module (alrmSimul.asm)
; Message strings
MENU     dc.b   "CEG3136 Alarm",NL,CR,"c-Config a-Arm",NL,CR,$00
NEWLINE  dc.b   NL,CR,$00
```

Symbolic constants and other definitions are found in include files (alarmSimul.h for C and alarmSimul.inc for assembler). Here are the contents of alarmSimul.inc:

```
NOCHAR        equ    0xff
BEL           equ    '@'
TRUE          equ    1
FALSE         equ    0
NUMCODES      equ    5
XWINSUM        equ   ('X'+'X'+'X')&$FF  ; sum of three X's
OWINSUM        equ   'O'+'O'+'O'        ; sum of three O's
NL             equ   $0a                ; NEWLINE
CR             equ   $0d                ; carriage return
ASCII_CONV_NUM equ   $30                ; For converting digit to ASCII
```

Alarm System Simulation Module

This module contains a main routine. The *main* subroutine C code is shown below. The code illustrates the following logic:

- Display menu.
- If 'c' is selected, call *configCodes()* from the Configuration Module.
- If 'a' is selected, call *enableAlarm()* from the Armed Module.
- If an invalid value is entered, print an error message, display menu, and prompt for another selection.
- Repeat forever.

```
/*-----------------------------------
 * Function: main
 * Description: The main routine for running the game.
 *              Initializes things (via initgb) and then
 *              loops forever playing games.
 * --------------------------------*/
void main(void)
{
   byte select;
   do
   {
      printf(MENU);
      printf(PROMPT);
      select = getchar();  // getchar is debug 12 routine
      putchar(select);  // echo character
      if(select == 'c') configCodes();
      else if(select == 'a') enableAlarm();
      else printf(ERRMSG);
   } while(1);  // loop forever
}
```

Configure Module

This module is composed of one principle subroutine *configCodes* and three supporting subroutines. C programs for all three subroutines are presented below.

In C (see config.c):

```
// Configuration Module
#define MSTCDMSG "Master code?\n"
#define CONFIGMSG "a:mstr 1-4:other\n"
#define CERRMSG "Bad entry\n"
#define GET_CODE_MSG "Code or 'd'\n"
#define ERR_MST_MSG "Cannot disable\n"
```

In Assembler (see config.asm):

```
// Configuration Module
MSTCDMSG    dc.b "Master code",NL,CR,$00
CONFIGMSG   dc.b "a:mstr 1-4:other",NL,CR,$00
CERRMSG     dc.b "Bad entry",NL,CR,$00
GET_CODE_MSG dc.b "Code or 'd'",NL,CR,$00
ERR_MST_MSG dc.b "Cannot disable",NL,CR,$00
```

```
/*---------------------
 * Function: configCodes
 * Parameters: none
 * Returns: nothing
 * Description: Gets user to select code to update/disable. Calls
 *              setcode to update the code.
 * ---------------------*/
void configCodes()
{
   byte ix;  // index into array
   byte input; // input from user.
   byte flag;

   if(enterMstCode())
     do
     {
        printf(CONFIGMSG);
        input = getchar();
        flag = TRUE;
        if(input == 'm') setcode(0);
        else if( input>='1' && input <= '4')
        {
          ix = input-ASCII_CONV_NUM;
          setcode(ix);
        }
        else
        {
           printf(CERRMSG);
           flag = FALSE;
        }
     } while(!flag);
}
```

```
/*--------------------------------
 * Function: enterMstCode
 * Parameters: none
 * Returns: TRUE - valid code entered, FALSE otherwise.
 * Description: Prompts user for the 4 digit master alarm code.
 *-------------------------------*/
byte enterMstCode(void)
{
   byte retval = FALSE;
   byte i;
   byte input;
   int alarmCode = 0;
   int mult = 1000;

   printf(MSTCDMSG);
   for(i=0 ; i<4 ; i++)
   {
      input = getchar();
      if(!isdigit(input)) break;
      else
      {
        alarmCode = alarmCode + mult*(input-ASCII_CONV_NUM);
        mult = mult/10;
        if(mult == 0)
        {
          if(alarmCode == alarmCodes[0]) retval = TRUE;
        }
      }
   }
   return(retval);
}
/*--------------------------------
 * Function: setcode
 * Parameters
 *         ix - index of alarm code to update
 * Description: Prompts user for a 4 digit alarm code to
 *              update the alarm code at index ix. If 'd'
 *              is entered the alarm code is disabled.  It
 *              is not allowed to diable the master
 *              alarm code.
 *-------------------------------*/
void setcode(byte ix)
{
   byte flag = TRUE;
   byte i;
   byte input;
   int digit;
   int alarmCode=0;
   int mult=1000; // multiplier

   do
   {
       printf(GET_CODE_MSG);
       for(i=0 ; i< 4 ; i++)
       {
          input = getchar();
          if(input == 'd')
          {
```

```
            if(ix == 0) printf(ERR_MST_MSG);
            else
            {
               alarmCode = 0xffff;
               flag = FALSE;
               break;
            }
         }
         else if(isdigit(input))
         {
              digit = input - ASCII_CONV_NUM;
              alarmCode = alarmCode + digit * mult;
              mult = mult/10;
              if(mult == 0) flag = FALSE;
         }
         else
         {
             printf(CERRMSG);
             break;
         }
      }
   } while(flag);
   writeToEE( (int) &alarmCodes[ix], alarmCode);
}
/*----------------------------
 * Function: writeToEE
 * Parameters
 *       address - address location in EEProm to store code
 *       code - 2-byte integer code.
 * Description:
 *     Write the alarm code in EEProm. writeEEByte() is a
 *     function avaible in Debug12.
 *---------------------------*/
void writeToEE(int address, int code)
{
    writeEEByte(address, (byte)((code&0xff00)>>8) );
    writeEEByte(address+1, (byte)(code&0xff) );
}
```

Armed Module

This module is responsible for arming the alarm system (after a valid code has been provided and after a delay of 10 sec), monitors the system (and triggers the alarm if a door/window is opened), and disarms the system if a valid code is entered. If the alarm is triggered, it can be turned off by entering a valid code.

In C (see armed.c):

```
// Armed Module (armed.c)
#define CODEMSG   "Code\n"
#define ARMING "*** Arming ***\n"
#define ARMED     "*** Armed ***\n"
#define DISARMING "-- Disarming --\n"
```

In Assembler (see armed.asm):

```
CODEMSG dc.b  "Code",NL,CR,$00
ARMING  dc.b  "*** Arming ***",NL,CR,$00
ARMED     dc.b  "*** Armed ***",NL,CR,$00
DISARMING dc.b  "-- Disarming --",NL,CR,$00
```

The principle subroutine is *enableAlarm()* that is called from the *main* function. This function will validate an alarm code. If valid, the function *monitor()* is called to monitor for either a trigger event (characters, a, b, c, d, or e) to trigger the alarm. In the case of the trigger event 'a', a delay of 10 seconds allow for disarming the alarm system before triggering the alarm. If an alarm is triggered, the bel characters ('@') are sent to the terminal (with delays between each @). Entering a code at any time disarms/turns off the alarm system.

The C functions for the subroutines in the Armed Module are shown below. Given that code must validated while the system is monitoring for triggering events and during an alarm (sending of bel characters to the terminal), the approach to validating a code must be done one character at a time. The calling routine is expected to call the routine regularly. Thus it must check for an entry from the user (using polling), if nothing has been entered, then return immediately with the value FALSE. If the user has entered a character, then process it as follows:

1) If it is a digit, update the variable *code* variable according to the current position (the current position of the digit is determined by the value of *mult*; see the subroutine *setcodes* for details). If it is the 4th digit, then the value of *code* is compared to all values in the codes array. If the code is valid, then TRUE is returned, otherwise FALSE is returned and variables (*code* and *mult*) are reset to read in another code.

2) If it a non numeric character, then the variables *code* and *mult* are reset to read in a new 4 digit code.

```
/*-------------------------------------------------------------
 * File:  armed.c
 * Description: This file contains the Amred module for the
 *              Alarm System Simulation project.
-----------------------------------------------------------------*/

#include "alarmSimulExtern.h"  // Definitions file

// Constants
#define ARMDELAY 10000  // delay for arming and disarming
#define BEEPDELAY 1000

// Armed Module (armed.c)
#define CODEMSG "Code\n"
#define ARMING "*** Arming ***\n"
#define ARMED   "*** Armed ***\n"
#define DISARMING     "-- Disarming --\n"

/*------------------------
 * Function: enableAlarm
 * Parameters: none
 * Returns: nothing
 * Description:
 *     Gets user to enter a valid alarm code to arm
 *     the alarm system.  Delays 15 seconds (to allow user
 *     to leave; the alarm can be disarmed during this period),
 *     monitors for trigger events (a, b, c, d, e).  When 'a'
 *     is detected, 10 seconds are allowed to enter an alarm code to
 *     disarm the system; otherwise the alarm is triggered. For other
 *     trigger events, the alarm is triggered instantly.  The alarm
 *     system can be turned off after being triggered with an alarm
 *     code.
 *------------------------*/
```

```
void enableAlarm(void)
{
   byte input;  // input from user
   byte codeValid;  // valid code found
   byte delayFlag;

   // Get a valid code to arm the system
   printf(CODEMSG);
   codeValid = FALSE;
   while(!codeValid)
   {
     input = getchar();
     codeValid = checkCode(input);
   }
   printf(ARMING);
   // Delay 10 seconds
   setDelay(ARMDELAY);
   codeValid = FALSE;
   delayFlag = FALSE;
   while(!delayFlag)
   {
      delayFlag = pollDelay();
      input = pollgetchar();  // check if alarm code entered to disarm
      if(isdigit(input) || input == '#')
      {
         codeValid = checkCode(input);
         if(codeValid) delayFlag = TRUE;  // break out of loop
      }
   }
   // Loop to monitor trigger events and alarm code to disable
   // codeValid is TRUE if valid alarm code entered during delay
   if(!codeValid) printf(ARMED);
   while(!codeValid)
   {
       input = pollgetchar();
       if(isdigit(input) || input == '#') codeValid = checkCode(input);
       else if(input == 'a') // Front door opened
       {
           printf(DISARMING);
           setDelay(ARMDELAY);
           while(!codeValid)
           {
              if(pollDelay() == TRUE)
              {
                 triggerAlarm();
                 codeValid = TRUE;
              }
              else
              {
                 input = pollgetchar(); // check if alarm code entered to disarm
                 if(isdigit(input) || input == '#')
                     codeValid = checkCode(input);
              }
           }
       }
       else if(input >='b' && input <= 'e') // other door/window opened
       {
           triggerAlarm();
```

```
            codeValid = TRUE;
        }
        // ignore all other input
    }
}
/*----------------------------
 * Functions: checkCode
 * Parameters: input - input character
 * Returns: TRUE - alarm code detected
 *          FALSE - alarm code not detected
 * Descriptions: Creates alarm code using digits entered until
 *               4 digits are seen. After 4th digit, see if
 *               alarm code is valid using isCodeValid().
 *----------------------------*/

byte checkCode(byte input)
{
   static int mult = 1000; // current multiplier of digit
   static int alarmCode = 0;   // alarm code value
   byte retval = FALSE;

   if(isdigit(input))
   {
      alarmCode = alarmCode + (input-ASCII_CONV_NUM)*mult;
      mult = mult/10;
      if(mult == 0)
      {
         retval = isCodeValid(alarmCode);
         alarmCode = 0;
         mult = 1000;
      }
   }
   else
   {
      alarmCode = 0;
      mult = 1000;
   }

   return(retval);
}

/*----------------------------
 * Functions: isCodeValid
 * Parameters: alarmCode - integer alarmCode
 * Returns: TRUE - alarm code valid
 *          FALSE - alarm code not valid
 * Descriptions: Checks to see if alarm code is in the
 *               alarmCodes array.
 *----------------------------*/

byte isCodeValid(int alarmCode)
{
   int *ptr;  // pointer to alarmCodes
   byte cnt = NUMCODES;
   byte retval = FALSE;
   ptr = alarmCodes;
   do
   {
```

```
      if(*ptr++ == alarmCode)
      {
         retval = TRUE;
         break;
      }
      cnt--;
   } while(cnt != 0);
   return(retval);
}

/*----------------------------
 * Functions: triggerAlarm
 * Parameters: none
 * Returns: nothing
 * Descriptions: Repeatedly sends a bel character to the
 *               terminal until a valid alarm code is entered.
 *----------------------------*/

void triggerAlarm()
{
   byte done = FALSE;
   byte doneInput;
   byte input;
   while(!done)
   {
       putchar(BEL);
       setDelay(BEEPDELAY);   // 1 sec between beeps
       doneInput = FALSE;
       while(!doneInput)
       {
          doneInput = pollDelay();
          input = pollgetchar();  // check if code entered to disarm
          if(isdigit(input) || input == '#') done = checkCode(input);
       }
   }
}
```

Delay Module

To be completed as part of Lab 1.

Utilities Module

This module contains utility routines. These are general routines that can be called by any of the other modules.

```
/*----------------------------------------
 * Function: isdigit
 * Parameters: none
 * Returns: TRUE - chr is a digit
 *          FALSE - chr is not a digit
 * Description: Returns TRUE if chr is a digit character
 *              and FALSE otherwise.
 *----------------------------------------*/
byte isdigit(char chr)
{
   byte retval = FALSE;  // return value
   if(chr >= '0' && chr <= '9') retval = TRUE;
   return(retval);
}

/*----------------------------------------
 * Function: pollgetchar
 * Parameters: none
 * Returns: NOCHAR - no character available for reading.
 *          a valide ASCII character read from SCI0
 * Description: Checks the RDRF bit to see if a character
 *              is available before reading a character.
 *----------------------------------------*/

char pollgetchar(void)
{
  char chr = NOCHAR;

  if(SCI0SR1_RDRF==1) chr = getchar();

  return(chr);
}
```