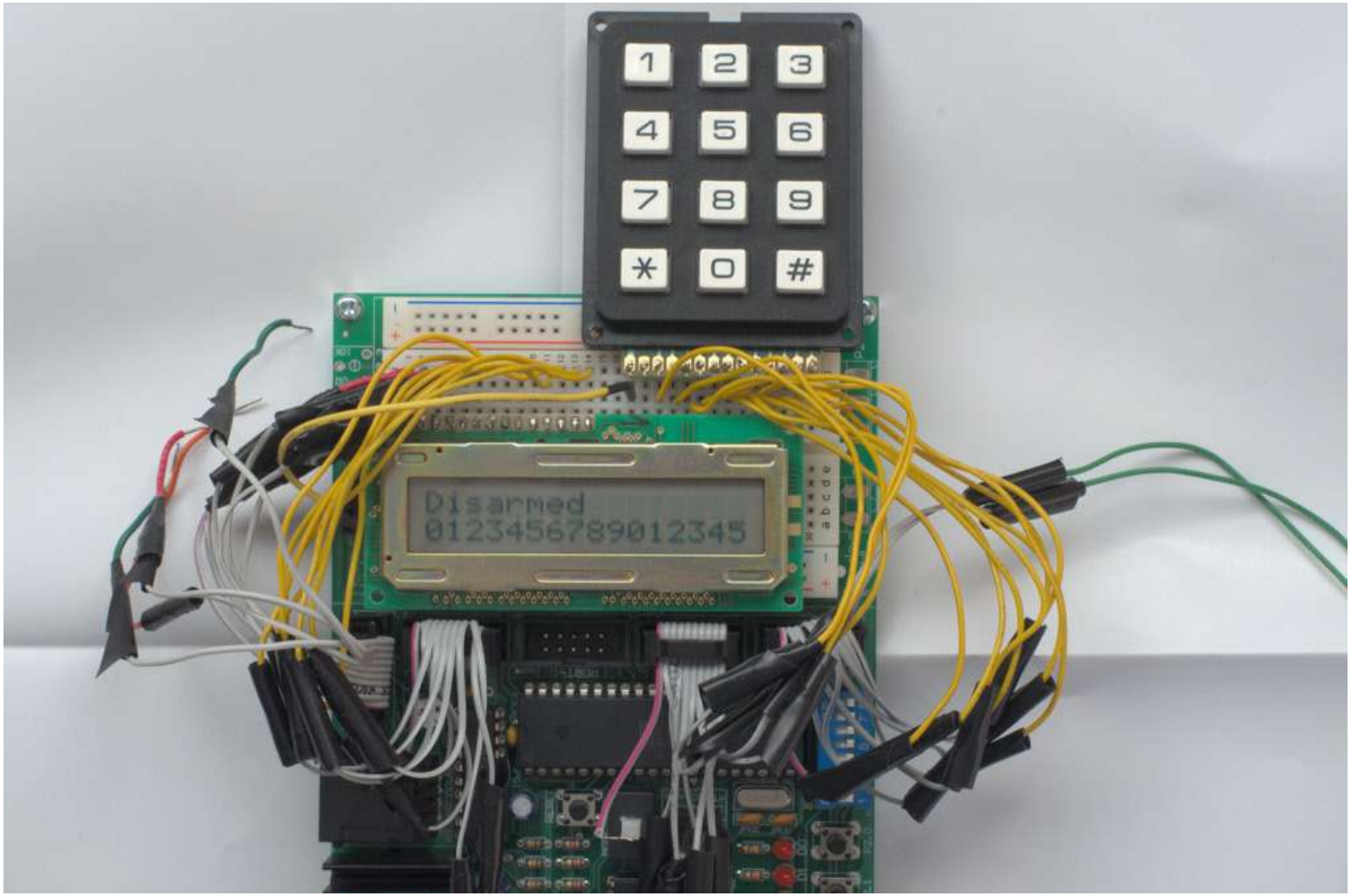
LCD display, C programming

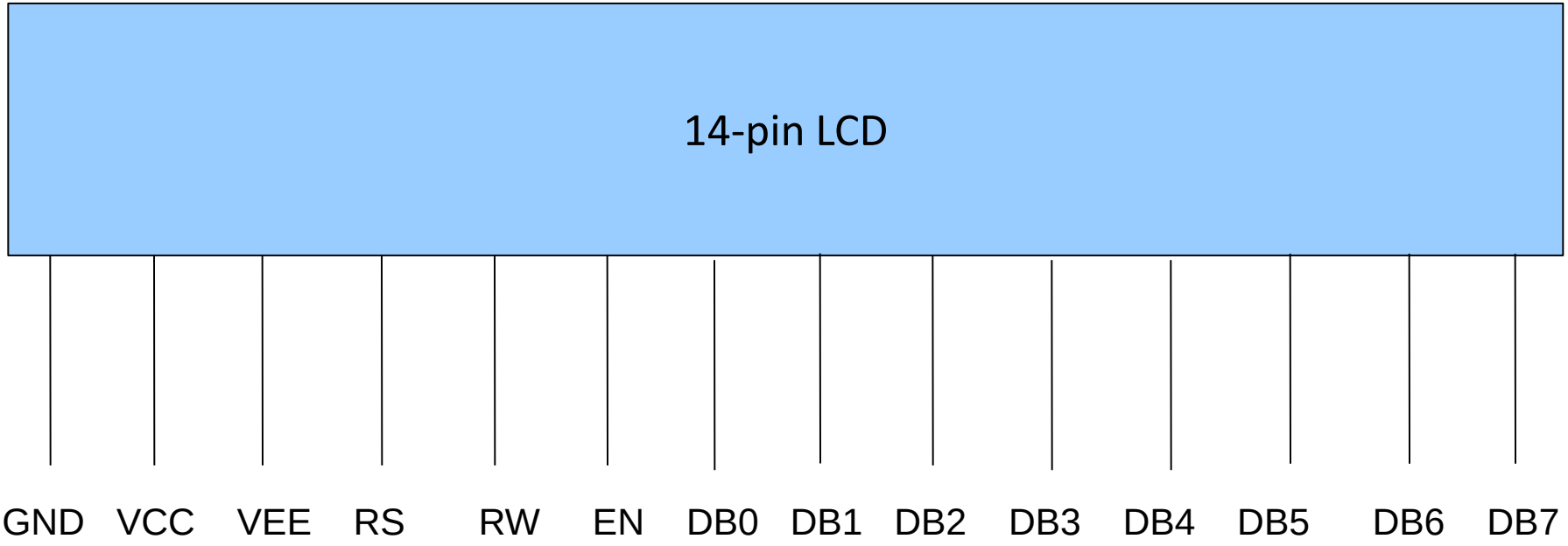


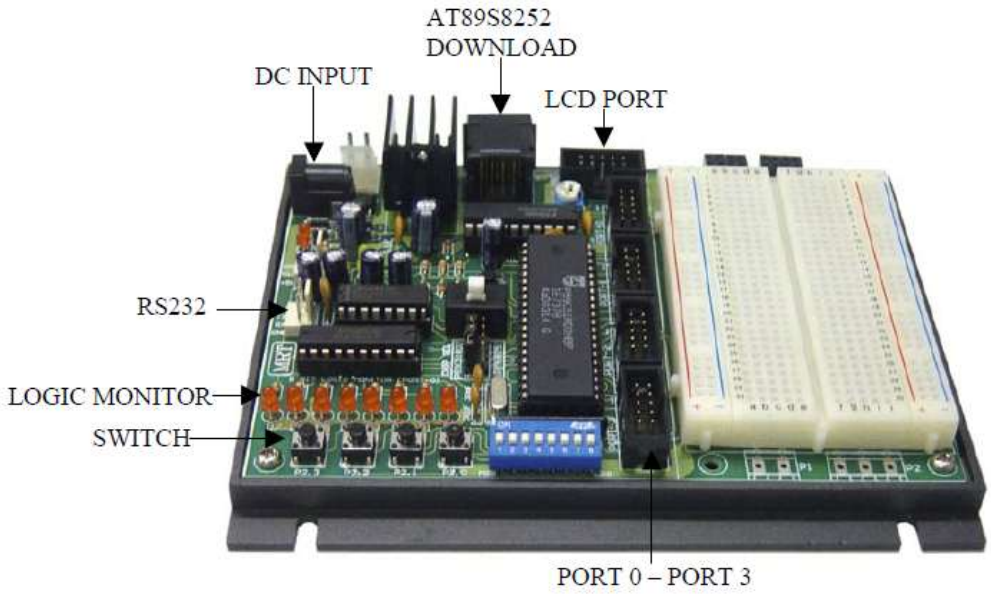
Overview

- LCD
  - 2 line text display
  - Physical connections
  - Control and data lines
- C programming
  - Organizing your code
  - Global variables
  - Parameter passing

LCD display

- LCD display:
  - Small
  - Low power
  - Simpler interface than serial communication to a monitor
  - 44780 LCD standard makes them mostly interchangeable
  - Displays 2 lines with 16 characters / line





PIN	NEME	Function connected with CPU
1	GND	GND
2	VCC	+VCC
3	VEE	VR Adjust the LCD light LCD
4	RS	P1.0 Instruction/Data Select
5	RW	P1.1 Read/Write Data
6	E	P1.2 Enable
7	D4	P1.4 Data Bit 4
8	D5	P1.5 Data Bit 5
9	D6	P1.6 Data Bit 6
10	D7	P1.7 Data Bit 7

Control lines

Line	Function
EN	Enable line Tells the LCD when it is being sent data or a command. Set EN=1, set other lines and data bus, then set EN=0, 1-0 transition causes LCD to read. EN must be high for minimum time (depends on particular make of LCD, ~250ns)
RS	Register select Tells the LCD if the information found on the data bus is a command or data for display. RS = 0, data is a command RS = 1, data is text for display
RW	Read/write Tells the LCD whether we are trying to read or write to the LCD. RW = 0, information on data bus is for writing to LCD RW = 1, for querying the LCD, e.g. checking if LCD busy

Data bus

- LCD can be set to either 4-bit or 8-bit mode
  - 8-bit easier, but requires more wires
    - Total of 11 data/control lines, +3 power/ground
    - Two cables:
      - LCD port for 3 power/ground, 3 control lines
      - Port P0, P2, or P3 for 8 data bus lines
  - 4-bit, must send command/text one nibble at a time
    - Total of 7 data/control lines, +3 power/ground
    - One cable:
      - LCD port for 3 power/ground, 3 control lines, 4 data bus lines
- Command or output text
  - Sent by placing 8-bit char value on data bus

Checking busy status

- Instructions take LCD time to process
  - LCD signals it is done by lowering level on DB7
  - Make a function that will be used by other LCD functions:
    - Specify a command, RS = 0
    - Specify we want to query LCD, RW = 1
    - Mark start of command, EN = 1
    - Set all pins on data bus to 1
    - Repeat process until DB7 is 0
    - Finish the command, EN = 0
    - Specify future commands will write to LCD, RW = 0

Checking busy status

```
WAIT_LCD:
    CLR EN                ; Start LCD command
    CLR RS                ; Specify an LCD command
    SETB RW               ; Specify we are reading from LCD
    MOV DATA,#0FFh        ; Set data bus to all 1's initially
    SETB EN               ; Signal LCD to process
    MOV A,DATA            ; Read the return value
    JB ACC.7,WAIT_LCD     ; If bit 7 high, LCD is still busy
    CLR EN                ; Finish the command
    CLR RW                ; Turn off RW for future commands
    RET
```

```
void LCDWait();
```

Issuing a command

- LCD accepts a variety of commands
  - Create a function that issues a command
  - Command is a byte on the input bus
  - Procedure (8-bit mode):
    - Set RS = 0 to indicate a command
    - Set RW = 0 to indicate a write
    - Move command onto data bus
    - Set EN = 1 to signal start of command
    - Wait 4 cycles
    - Set EN = 0 to mark end of command
    - Wait while LCD is busy

Sending a command

```
LCD_COMMAND:
    CLR RS              ; Specify this is a command
    CLR RW              ; Specify that we are writing
    MOV DATA,A          ; Put the command on the data bus
    SETB EN             ; Clock out command to LCD
    NOP                 ; Wait 4 cycles to give LCD time to process
    NOP
    NOP
    NOP
    CLR EN              ; Finish the command
    CALL WAIT_LCD       ; Wait for command to execute
    RET
```

```
void LCDSendCommand(unsigned char cmd);
```

Initialization commands

- Initializing the LCD, issue three commands:
  - 0x38 = 8-bit data bus, 5x8 character font
    - 0x20 = data interface command
    - 0x10 = bus size, 8-bit (otherwise 4-bit)
    - 0x08 = 2 lines LCD display (other 1-line)
    - 0x04 = character size 5x10 (otherwise 5x8)
  - 0x0C = turn on, with no cursor
    - 0x08 = display cursor command
    - 0x04 = display on (otherwise off)
    - 0x02 = cursor on (otherwise no cursor displayed)
    - 0x01 = cursor blinks (otherwise cursor constant)
  - 0x06 = turn on cursor auto-advance
    - 0x04 = cursor move direction command
    - 0x02 = advance cursor after write (otherwise don't)
    - 0x01 = shift display after write (otherwise don't)

```
void LCDInit();
```

Cursor position

- Clearing screen
  - Issue command 0x01

```
void LCDClear();
```

- Cursor position
  - Issue command:
    - 0x80 + desired location
  - Only the blue spots are visible

```
void LCDSetCursor(unsigned char line,
                  unsigned char index);
```

Display	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16
Line 1	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	...
Line 2	40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F	50	51	52	53	54	55	...

Writing a character

- Writing a single character
  - Will be placed at current cursor position
  - Procedure (8-bit mode):
    - Set RS = 1 to indicate text data instead of command
    - Set RW = 0 to indicate write operation
    - Move 8-bit char value to data bus
    - Set EN = 1 to mark start of command
    - Wait 4 cycles
    - Set EN = 0 to mark end of command
    - Wait while LCD busy

Writing a character

```
WRITE_LCD_TEXT:
   SETB RS              ; Specify this is text for display
   CLR RW               ; Specify that we are writing
   MOV DATA,A           ; Put the command on the data bus
   SETB EN              ; Clock out command to LCD
   NOP                  ; Wait 4 cycles to give LCD time to process
   NOP
   NOP
   NOP
   CLR EN               ; Finish the command
   CALL WAIT_LCD        ; Wait for command to execute
   RET
```

```
void LCDWriteChar(unsigned char ch);
void LCDWriteText(const char* str);
void LCDBlankLine(unsigned char line);
```

Code organization

- Option 1: Put everything in one giant main function
  - Code reuse: virtually impossible
    - Frequent repeated code that must be kept in synch
    - Using code in another project requires time and care
  - Bug density: extremely high
    - All variables are available to all parts of the code
    - Can't effectively test individual parts in isolation
    - High levels of nesting make it hard to see what is going on
  - Ability to find things: extremely low
    - No separation into functional parts

Code organization

- Option 2: Everything in one *.c file, use functions but pass data via global variables
  - Code reuse: tedious
    - Requires cutting out just the functions, constants, and globals related to the functionality you are moving
  - Bug density: high
    - Global variables lead to unforeseen dependencies
    - Bugs become harder to find and more squirrely
    - Functions have implicit dependency on 0+ global variables but this is not explicitly obvious from function parameter list
  - Ability to find things: okay
    - Need to find where the desired function is
    - No real order of the functions in what becomes a very long file

Code organization

- Option 3: Everything in one *.c file, use functions and avoid global variables
  - Code reuse: somewhat tedious
    - Requires cutting out just the functions and constants related to the functionality you are moving
  - Bug density: moderate
    - Functions do one simple job given their input parameters
    - If globals are required, they are accessible everywhere
  - Ability to find things: okay
    - Need to find where the desired function is
    - No real order of the functions in what becomes a very long file

Code organization

- Option 4: Separate different functionality into different *.h and *.c files
  - Code reuse: good
    - New project can just add the relevant pair of *.h and *.c files
  - Bug density: low
    - Functions do one simple job given their input parameters
    - If globals are needed, they can be isolated to their *.c file
      - static globals = private instance variables
      - static functions = private methods
  - Ability to find things: good
    - Look in relevant *.h file to see what functions are available
    - Look in relevant *.c file to see implementation of a function

sleep.h

```
// Power savings based sleep function that uses timer0
// Includes the timer0 type 1 interrupt function.

#ifndef __SLEEP_H__
#define __SLEEP_H__

#include <REG52.h>

void startTimer0(); // Start the 0.01s heartbeat on timer0
void sleep();       // Power savings sleep for 0.01s

// Sleep for the given number of hundreths of a second
void sleepHundreths(unsigned char hundreths);

#endif
```

sleep.c

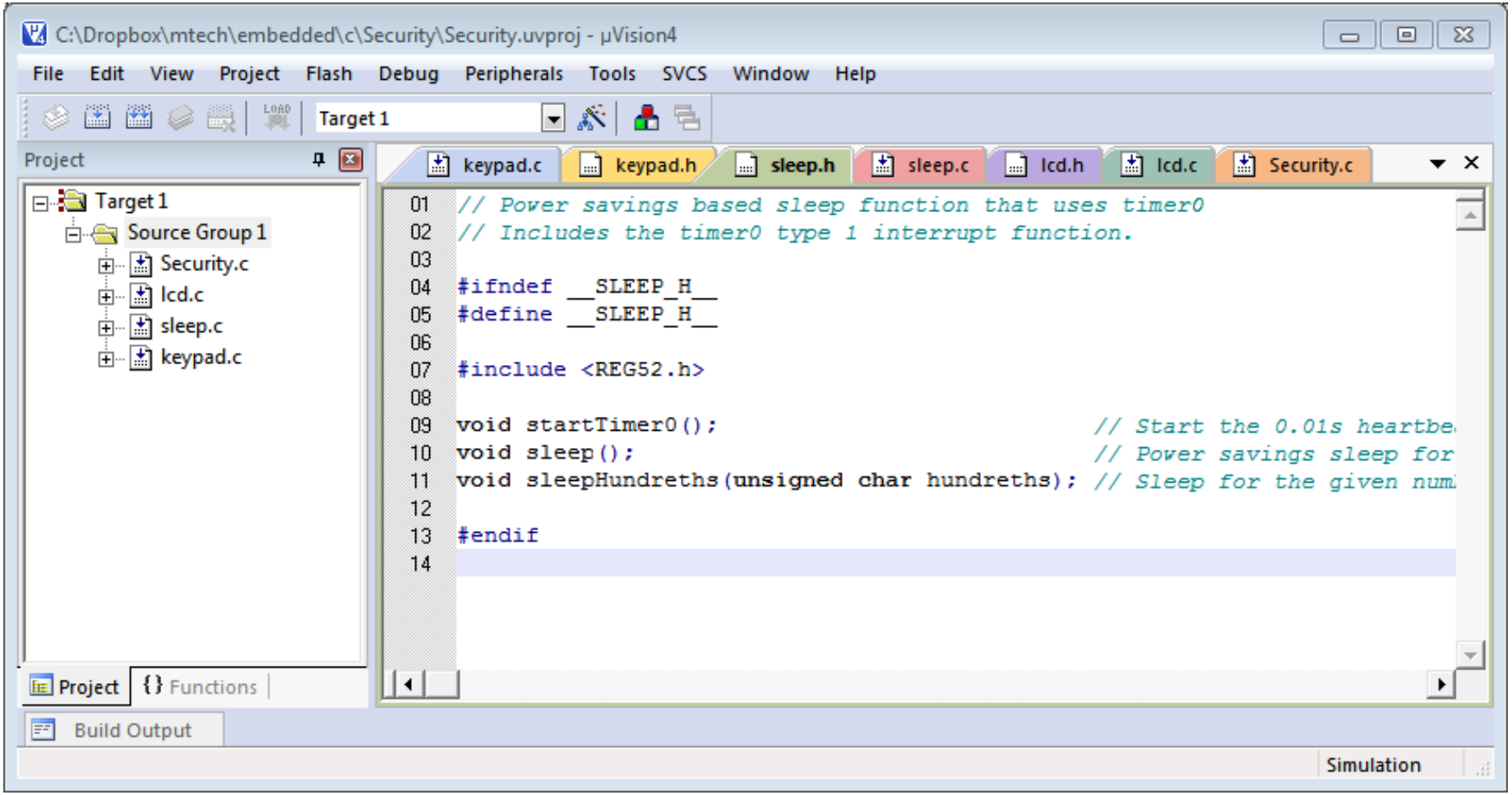
```
#include "sleep.h"

void startTimer0()
{
   TMOD = TMOD & 252; // Mask out lowest 2 bits
   TMOD++;            // Set T0M1/T0M0 to 01
   TH0  = 219;        // Setup for ~0.01 delay
   TR0  = 1;          // Start timer0 running
   ET0  = 1;          // Enable timer0 interrupt
   EA   = 1;          // Global interrupt enable
}

void timer0ISR() interrupt 1
{
   TH0  = 219;
}

void sleep()
{
   TH0  = 219;
   PCON = 1;
}

void sleepHundreths(unsigned char hundreths)
{
   unsigned char i = 0;
   for (i = 0; i < hundreths; i++)
      sleep();
}
```


C:\Dropbox\mtech\embedded\c\Security\Security.uvproj - µVision4
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
Target 1
Project
Target 1
Source Group 1
Security.c
lcd.c
sleep.c
keypad.c
keypad.c
keypad.h
sleep.h
sleep.c
lcd.h
lcd.c
Security.c
01 // Power savings based sleep function that uses timer0
02 // Includes the timer0 type 1 interrupt function.
03
04 #ifndef __SLEEP_H__
05 #define __SLEEP_H__
06
07 #include <REG52.h>
08
09 void startTimer0(); // Start the 0.01s heartbe
10 void sleep(); // Power savings sleep for
11 void sleepHundreths(unsigned char hundreths); // Sleep for the given num
12
13 #endif
14
Project
Functions
Build Output
Simulation