EE342 Microcontrollers
Initial Lab Exercise: Terminal Program

Introduction:

The purpose of this document is to describe the initial exercise basis for projects to be done during the first part of EE342. The idea is to use the PC as a “terminal” (something you can type into and receive and display text) to operate an embedded program on the microcontroller that presents to the user a “command line interface (CLI). This program executes different “commands” that the user types into the terminal program on the PC, and sends back information to be displayed. We use a serial port (which you have already studied in EE247) for the connection between the PC and the microcontroller.

The “terminal” module is “stackware” or “middleware” that came with the DEMOJM boards. It is general in its usefulness, which is what you want from “middleware” that fits between the user application and the device specific drivers that interact with the hardware. For what we are doing, we are using the “terminal” module with a different set of drivers than would be used with the JM family devices.

This document is intended to include information needed to get the terminal program up and running on the S08SH8 processors we will be using. (The same also works with almost any other processor, say, the QG8 from Mechatronics, though perhaps with modification of the drivers.)

Overview:

You will often see diagrams that show the relationship among different modules running in a computer. Usually these are shown with hardware levels at the bottom and the user’s application at the top. The applicable diagram for this project is shown in Figure 1. The “terminal” module lies where you would usually find an operating system. Indeed, the terminal can be thought of as a very basic operating system. It receives commands from the user and causes appropriate applications to run, and sends appropriate data back to the user.

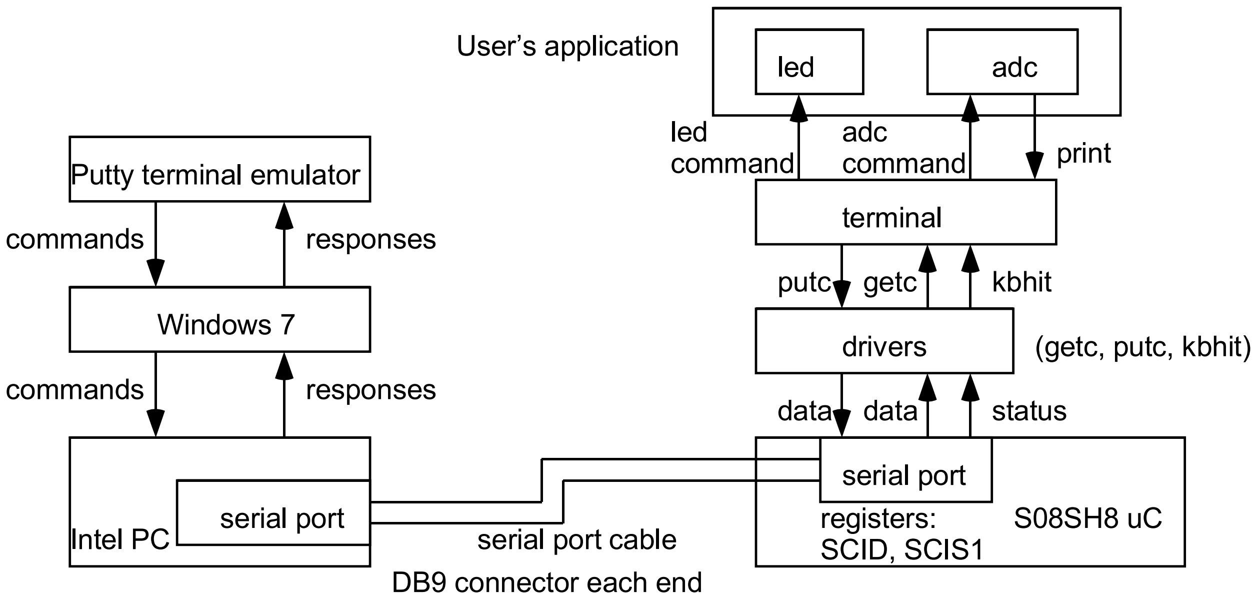

Figure 1 Terminal Program Context

So, for us, we will end up with the following modules that get compiled together to make up the executable for the microcontroller:

1. Main (main.c): main program including initialization, the "main loop" and the application commands (Often we'd separate out initialization and application commands).

2. The Terminal module (hcc_terminal.c, hccterminal.h, utils.c, utils.h) : (You could consider Utils (format conversion utilities) a separate module. We'll lump them in here.)

3. The drivers for the serial port (and related functions): SCI_Functions.c, SCI_functions.h

Other files: We need definitions from mc9s08sh8.h, hcc_types.h, derivative.h for the modules.

Getting it Made:

Code Warrior 10 is picky about its "workspace". It does not like you to copy a project in from somewhere else. It complains. Sometimes it works anyway, but it does complain and there may be some problems down the line. So, we will build this project as if from scratch, but copy in the needed material from files provided (as listed above). So, bring up CodeWarrior 10, and choose where you want your "Workspace" to be. (You could make it a thumb drive, your H drive, or even put it in your user folder on the particular machine's C drive. If you do that, make a copy to a thumb drive, but don't expect CodeWarrior to like it if you try to use it directly.)

Create a new project (I called mine "shterminal") for the S08SH8 processor. You can take the defaults. (You could decide to include floating point; if so make it 32 bit floats.) We will use C, NOT C++. (If you choose C++, a very tight memory restriction is imposed.) Ultimately you end up with a default project with a main and some header files.

Now, add to your project the header files (into the "headers" folder) code files (into the "sources" folder). Be sure they get added to the project, and show up correctly in the project navigator. There will be a default "main". We will want to replace that with the main for this project. You can either replace the file, or just replace the file contents. It should NOT be necessary to move mc9s08sh8.h into the project; that should already be there. It's possible derivative.h will already be there too; I don't remember.

So, with all that done, you should be able to make the project and debug.

Running It:

So, "debug" the (successfully) compiled project. That downloads the code into the microcontroller and sets it up ready to run.

Now, connect the microcontroller to the PC using the serial port cable. Check the strappings on the microcontroller board to be sure that the serial port is enabled.

Next, start PuTTy on the PC. You will get a configuration window. You need to designate use of the "serial" port. You also need to indicate the bit rate: we will use 115200 bits per second. (You sometimes hear this called baud rate.) Once you hit return you should have a blank screen.

It's time to start the microcontroller executing. You should see an initialization message appear on the screen. Then there's a return, and you should see the ">", a prompt symbol, on the left. This is prompting you to give a command. Try typing "help". When you hit return, you should see a response that lists the various commands: help, led, and adc. (Later we'll add more.) try the "led" command. When you hit return, the screen will show another prompt, and an LED on the microcontroller board should flip state between on and off. If you do it again, it flips back.

The "adc" command needs a "parameter". You type "adc 1" top get the potentiometer Voltage, and "adc 2" for the light sensor Voltage. If you type just "adc", the terminal program crashes. You can twiddle the pot or put your hand over the board to shade it to vary the illumination. You get a response to the screen. Is it correct? (It isn't! But, it does vary.)

So, the terminal program is working. It needs some fixing though. For one thing, we'd like correct Voltages. For second, we'd like the adc command not to crash the machine if we forget the parameter. Maybe we'd like to be able to flip either LED. Maybe we'd like an alarm clock that we can set. But to do that, we need to understand how it all works.

The Drivers

Given that you have already studied the serial port, the driver functions should be pretty easily understood. When we initialize the terminal (inside the microcontroller) we need to do several things. We need to be sure we have the microcontroller system clock running at the right speed. We need to initialize the port for the correct bit rate, and turn on the transmitter and receiver. The three functions we need are:

int putch(char), int getch(void), and int kbhit(void).

(You will see these called putc, getc, and various other variations on those names.)

The three functions put a character into the serial port, get a character out of the serial port, and check to see if there is a character waiting to be picked up. An oddity of these functions is that they all return integers (ints), not characters. This is an oddity that goes all the way back to the origins of the C language; it's how these functions have been defined as long as C has been around, since forever. What that means is that we will have to use "casts" to change characters into ints and ints into characters.

Take a looks at all of these functions. Here I will walk through just putch. The function is given below in Figure 2.

```
/*void TERMIO_PutChar(char send)*/
int TERMIO_PutChar(char send){
  int dummy;
  while(!SCIS1_TDRE);/* wait until TDRE=1 */
  dummy = SCIS1;
  SCID  = send;
  dummy=send;
  return dummy; /*added line*/
}//end SPI_PutChar
```

Figure 2 putch function

This is a "synchronous" function. It won't return until its job is done. (An "asynchronous" function would get something started, then return immediately. The function putch checks to see if the transmit Data Register Empty (TDRE) flag is set. If it is, the data "send" is stuffed into the serial port data register and the function returns the same data "send" that was sent (but as an integer). That's why the integer "dummy" is used. If TDRE is NOT set, then it just does a spin wait. It could be waiting a looooooong time! At least until whatever character is in the transmit buffer to go out. At 115200 bps that could take as much as 87uS. At 9600 bps (an often-used standard speed) that would take 1.04 mS. That's a long time.

Notice that these particular function names are TERMIO_PutChar, TERMIO_GetChar, and TERMIO_Kbhit. That's to distinguish them from other potential putch, getch, and kbhit functions for other kinds of interfaces, say, via USB, which use different registers and code to interact with the device hardware. So, these are the drivers specific to the serial port on an 'SH8 microcontroller. The terminal module itself is NOT specific to this kind of serial port. In like manner, the initializations are unique to the SH8. That's why we gather all this SH8 serial port specific stuff into this one "SCI_Functions module (file pair).

The main program needs to call any functions to initialize the serial port before initializing the terminal. So in main, we find the following code fragment as shown in Figure 3 below. After the parallel port to the LED is initialized, the clock and the serial port initialization calls are made. Then the call to initialize the terminal can be made, and here is where we tell the terminal what the functions are to use for putch, getch, and kbhit.

```
PTBDD=0x80;
ICS_FEI_16M();
TERMIO_Init_16M_115200();
terminal_init(TERMIO_PutChar, TERMIO_GetChar, TERMIO_kbhit);
```

Figure 3 SCI Initialization (in main)

Terminal Initialization of Drivers:

Take a look at the file hcc_terminal.c. Near the top you will find several global variables defined. Three of them are shown (an excerpt form the code) in Figure 4 below:

```
static int (*putch)(char);
static int (*getch)(void);
static int (*kbhit)(void);
```

Figure 4 terminal driver function variables

What these three statements do is create three variables which will be known as pointers to functions called (within the terminal module) putch, getch, and kbhit. They are of integer (16 bit) size. When the function terminal_init is called, three pointers are passed, and those pointers are put into these three variables. (The names used in the function declaration have to be different, since the arguments of the function are passed as stack variables, and we need those values, the pointers to the three functions, to go into global variables.) See Figure 5 below.

```
void terminal_init(int (*putch_)(char), int (*getch_)(void), int(*kbhit_)(void))
{
  cmd_line[sizeof(cmd_line)-1]='\0';
  cmd_line_ndx=0;

  cmds[0]=(void *)&help_cmd;
  n_cmd=1;

  putch=putch_;
  getch=getch_;
  kbhit=kbhit_;

  print_greeting();
  print_prompt();
```

Figure 5 Terminal Initialization function.

Commands and Help:

The terminal works by receiving characters from the user via the serial port, and then doing something appropriate to those commands. So, before we go further, we need to know how a "command" is represented. In terms of C, each command is an instance of a struct called a "command_t". Because commands need to be defined by the user (in other modules) as well as used by the terminal itself, the definition of a "command_t" needs to be "published" by putting it in the file "hcc_terminal.h", not just "hcc_terminal.c". Here's the definition (Figure 6):

```
typedef struct {
  const char *txt;
  cmd_func * func;
  const char *help_txt;
} command_t;
```

Figure 6 Command structure

So, a command has three elements. There is a pointer to a string of text called "*txt". The "const" says that the character data itself is constant; it is stored in FLASH memory (essentially EPROM) and can't be changed. The pointer CAN be changed, because the structure itself is going to be stored in RAM. It's a variable. In fact, there is an array of pointers to these command_t structs called "cmds" (commands). It is declared to exist in the statement:

```
static command_t * cmds[MAX_CMDS];
```

This statement says there is in global variables (and hence static) an array variable "cmds" which consists of some number of pointers to command_t's. So, if MAX_CMDS is defined as 10, there are 10 pointers (of 16 bits each) in this array. Initially the array is empty. There's a global variable n_cmd that tells us how many commands there are, and it starts out undefined. (Actually, yes, this variable is global since any function that knows about it can access it. It is static, meaning it doesn't go away when you are outside some function; it's persistent. But since it is not in hcc_terminal.h, nobody else knows about it in other modules.)

```
static hcc_u8 n_cmd;
```

So, where are the actual commands that cmds points to? Well, the first one is right here in hcc_terminal.c. It's one of the module variables. Figure 7 shows the "Help" command.

```
static const command_t help_cmd = {
  "help", cmd_help, "Prints help about commands. "
};
```

Figure 7 The Help Command

We saw that the struct for a command_t has three elements. This statement says there is a static global variable called "help_cmd" that is a pointer to a "command_t", the command structure. Furthermore, the code gives a value for this structure (with the "= {….}" part of the statement. The element "txt" will point to a string that reads "help". The element func points to a command function (defined earlier) called "cmd_help". The third variable, "help_txt", points to a string that describes what the help command does. The definition of a cmd_help is earlier:

```
typedef void (cmd_func)(char *params);
```

(from hcc_terminal.h)

So, here again, we have a pointer to a function. This one, generically a "cmd_func" (that's what we call the type of function in this typedef), returns nothing (void) and is passed a string (a pointer to characters) called "params". The name of one of these command functions is "help_cmd", and we can find it right here in the hcc_terminal file (Figure 8). It should be pretty easy to see what this does. It just goes down the list of commands printing the strings it finds.

```
static void cmd_help(char *param)
{
  int x;

  param++;
  print("I understand the following commands:\r\n");

  for(x=0; x < n_cmd; x++)
  {
    print("  ");
    print((char *)cmds[x]->txt);
    print(":\t");
    print((char *)cmds[x]->help_txt);
    print("\r\n");
  }
  print("\r\n");
}
```

Figure 8 Help Command function

You might be wondering about the "params++;" statement. Because this is a command function, there is a pointer that is passed to it. But, the help command does not need a pointer. But if params was never used in the function, the compiler would generate a warning saying that there is an unused variable. By adding 1 to params we touch the variable just to suppress the warning. (We would not do this if we were low on memory, specifically FLASH memory. But we have 8K. Another byte or two for this line of code won't kill us.)

So, the first thing terminal_init does is install the help command by pointing the first element of cmds to it, then setting the number of commands to 1. Well, not quite the first thing. The first thing it does is initialize the array where the serial port input will be stored, the array "cmd_line". As characters come in, this is where they will go until the terminal recognizes that the user has hit return.

```
cmd_line[sizeof(cmd_line)-1]='\0';
```

Other commands are added by the function: `int terminal_add_cmd(command_t *cmd)`. If you look at the next 2 lines of main after terminal initialization, you will see how the commands "led" and "adc" are added. So, with that, the terminal is ready to run. We enter the main loop.

```
  (void)terminal_add_cmd((command_t*)&led_cmd);
  (void)terminal_add_cmd((command_t*)&adc_cmd);
```

Terminal Operation:

So, things are set up. What next? As soon as the receiver is turned on the serial port can start receiving characters. But, nothing happens until program execution asks about and picks up those characters. How does that happen? This is a "polled" program. Interrupts are enabled, but there is no interrupt code in the executable code, so the only way something happens is by normal program execution. So, as the main loop executes repearedly, we find a terminal call:

The main loop is shown in Figure 9 below:

```
for(;;) {
  c=SCIS1;
  terminal_process(); /* services terminal, dispatches commands*/
  DisableInterrupts;
  __RESET_WATCHDOG(); /* feeds the dog */
  EnableInterrupts;
} /* loop forever */
```

Figure 9 main Loop

The reason for the "c=SCIS1" statement is to let the debugger conveniently show the status of the serial port as variable c. It's really not needed. It's the next statement that is key: a call to the function "terminal_process" that actually does the work of the terminal. The function terminal_process checks to see if a character has come in to the serial port. The code for the first part of the function is shown is shown in Figure 10.

```
void terminal_process(void)
{
  char c;    /* moved up one line */
  while(c=(*kbhit)())
  {
    c=(char)(*getch)();
    /*while(c!=(char)(*putch)(c));*/
    /* Replace with just a simple call to putch()*/
    c=(char)(*putch)(c);
    if (c=='\r')
    {
      while('\n'!=(char)(*putch)('\n'))
        ;
    }
    /* Execute command if enter is received, or cmd_line is full. */
    if ((c=='\r') || (cmd_line_ndx == sizeof(cmd_line)-2))
    {
```

Figure 10 Terminal process and Character pickup

Notice that terminal_process does not communicate with main(); it neither gets arguments nor returns a value. It just happens. The first thing it does is call kbhit to pick up a character called c. Notice the notation for calling a function that is actually pointed to by a variable. It's a bit tricky. (We'll see that again.) There is no calling argument; really all we want to know is whether there's a character to receive. If there is not, we fall out of the while loop and return from terminal_process. If there is, we stay in terminal_process until we run out of characters to pick up. Remember, terminal_process does not know that characters are coming from a serial port one per 84uS at most, and there will never be more than one at a time. There could be a whole buffer full in other circumstances. That's why the while loop; we need to keep doing terminal stuff until we are done; other stuff in the main loop will just have to wait.

If there is a character, we pick it up with getch. Whatever the function was that was passed in during terminal initialization. The character actually comes back as an int, so it must be "cast" as a character for the character variable c, which is where we want to put it. Now, we immediately take the same character and put it back out by calling the putc function.

As the original terminal program had it, the call to putch was in the form:

```
while(c!=(char)(*putch)(c));
```

What that does is keep calling putch until it returns the same value that was passed to it. A non-blocking (asynchronous) version of putch might look and say, "TDRE is 0; we can't send the character out." It would then return a -1 (which is a legitimate int; it's not a legitimate char). Or something else not equal to the character. If that happens, terminal_process turns right around and keeps calling until it gets what it wants, the same character returned as it wanted to send, which indicates that putch has accepted the character and it will eventually go out. Now, in this case putch is doing the same thing; it's checking TDRE to make sure the character can be sent, so it will never return unless it's returning the same value. That's the reason for the simplification, to remove the while loop. (There's a perverse case in which I've seen this thing hang up.)

Next, a test is made to see if the character received is '\r', the "return" character. (we also add a '\n' for new line after the return; otherwise the next line on the terminal will overwrite the line just received.) If the character is not '\r', then we are done; terminal_process returns and the main loop continues.

If a '\r' is received, then terminal process must look down the list of commands it has to see if the first part of the string of characters just receives matches anything. If not, it tells the user that the command is not understood. If a match is found, it calls the corresponding function out of the command_t for that command. When the command function completes, it returns here to terminal_process, which then keeps receiving characters or returns if no more characters have come in. See if you can follow the logic of the remainder of terminal_process. What you will find is that the parameter passed to the command function, "param", points to the part of the command string just past the command itself. So, if you type "adc 1", then return, the pointer param will point to "1". (Not the numerical value 1, but a string consisting of the asci characters '1' and '\0' , that is 0x31, 0x00.)

User Commands:

The real guts of the user's application consists of the command functions which do the business that needs to get done. In this simple example, there are two of them, "led" and "adc". The "led" command function (from main.c) is given in Figure 11 below.

```
/* command called function to modify LED state */
static void cmd_led(char *param)
{
  static led_on=0;
  param++;
  if (led_on)
  {
    led_on=0;
    PTBD_PTBD7=0;
  }
  else
  {
    led_on=1;
    PTBD_PTBD7=1;
  }
}
```

Figure 11 Toggle the Led Function

Note that here again the parameter "param" is unused. The code just flips the bit at PTBD_PTBD7. (Really good code practice would abstract this into a macro or function.) That's it! There really isn't anything else to do!

The “adc” command is more complicated. It does use param to tell which channel of the A/D converter to collect. See Figure 12.

```
/* function to report ADC results */
static void cmd_adc(char *param){
  static channel=0;
  int i;
  char s[4];
  ADCCFG=0x40; /*10 bit mode*/
  for(i=0;i<10&&param[i]!=0;i++){
    if((param[i]=='1')||(param[i]=='2'))break;
  }
  if(param[i]=='1'){
    /* collect pot value */
    APCTL1=0x01;
    ADCSC1=0;
  } else if(param[i]=='2'){
    /* collect photo value */
    APCTL1=0x02;
    ADCSC1=1;
  }
  while(ADCSC1_COCO!=1);
  i=ADCR;
  s[0]=((i*5)>>10);
  s[1]='.' ;
  i=i-(s[0]<<10);
  s[0]=s[0]+'0';
  s[2]=((i*50)>>10)+'0';
  s[3]=0;
  print(s);
}
```

Figure 12 Analog data Function

Notice that the basic approach is to look at the parameter passed in from the terminal and see if it is a ‘1’ or a ‘2’ (again, the asci character, not the value). It can be a bit tricky because the user might type in an extra space before the 1 or the 2. Uh oh. Or, what if the user didn’t put a ‘1’ or a ‘2’? Running the A/D converter is straightforward (as done for EGR222). Converting that value to a character string is a challenge. The current code doesn’t work! Can you fix it? Hint: the utils.c file contains functions that change integers into strings. But, how do you get an integer for a fractional Voltage? (Second hint: change it to mV then move the decimal). So, see if you can fix this function.

Conclusions:

This document should help you with understanding the terminal program. If you do understand all this, you are doing very well! If not, keep on plugging. Add some more commands. We are going to want to add a clock using one of the device timers. How do you read out the time (from variables containing hours, minutes, and seconds)? How do you set the time? Those will be two commands. Later we will be adding commands to set track Voltages and flip switches. We’ll want to be able to monitor track Voltages too. We may want to make receiving characters interrupt driven so that stuff going on in the main loop can’t interfere. We may want to buffer outputs and make that interrupt driven as well so that we don’t tie up the microcontroller in busy-wait mode while sending long messages with slow characters at 9600 bps. Lots of ways to go from here!

Appendix A Code files:

Headers:

derivative.h:

```
/*
 * Note: This file is recreated by the project wizard whenever the MCU is
 *       changed and should not be edited by hand
 */

/* Include the derivative-specific header file */
#include <MC9S08SH8.h>

#define _Stop asm ( stop; )
  /*!< Macro to enter stop modes, STOPE bit in SOPT1 register must be set prior to
executing this macro */

#define _Wait asm ( wait; )
  /*!< Macro to enter wait mode */
```

hcc_types.h:

```
/****************************************************************************
 *
 *             Copyright (c) 2006-2007 by CMX Systems, Inc.
 *
 * This software is copyrighted by and is the sole property of
 * CMX.  All rights, title, ownership, or other interests0
 * in the software remain the property of CMX.  This
 * software may only be used in accordance with the corresponding
 * license agreement.  Any unauthorized use, duplication, transmission,
 * distribution, or disclosure of this software is expressly forbidden.
 *
 * This Copyright notice may not be removed or modified without prior
 * written consent of CMX.
 *
 * CMX reserves the right to modify this software without notice.
 *
 * CMX Systems, Inc.
 * 12276 San Jose Blvd. #511
 * Jacksonville, FL 32223
 * USA
 *
 * Tel:  (904) 880-1840
 * Fax:  (904) 880-1632
 * http: www.cmx.com
 * email: cmx@cmx.com
 *
 ***************************************************************************/
#ifndef _CMX_TYPES_H_
#define _CMX_TYPES_H_

/* Type definitions */
typedef unsigned char hcc_u8;
typedef unsigned int hcc_u16;
typedef unsigned long hcc_u32;

typedef volatile hcc_u8 hcc_reg8;
typedef volatile hcc_u16 hcc_reg16;
typedef volatile hcc_u32 hcc_reg32;

typedef hcc_u8 hcc_imask;

#ifdef NDEBUG
```

```
#define CMX_ASSERT(c) (void)0
#else
#define CMX_ASSERT(c)\
do {\
  if(!(c))\
  {\
    int a=1;\
    while(a)\
      ;\
  }\
}while(0)
#endif

#define BREW32(v)   ((hcc_u32)(((hcc_u32)(((hcc_u32)(v)) << 24)) \
                     | ((hcc_u32)(((hcc_u32)(v)) >> 24)) \
                     | (hcc_u32)((hcc_u32)((hcc_u32)(v) & (hcc_u32)0xff00ul) << 8)\
                     | (hcc_u32)((hcc_u32)((hcc_u32)(v) & (hcc_u32)0xff0000ul) >> 8)))

#define BREW16(v)   ((hcc_u16)(((hcc_u16)(v)) << 8) | (hcc_u16)(((hcc_u16)(v)) >> 8))

#define WR_LE32(a, v) ((*(hcc_u32*)(a))=BREW32(v))
#define WR_LE16(a, v) ((*(hcc_u16*)(a))=BREW16(v))
#define RD_LE32(a)    (BREW32(*((hcc_u32*)(a))))
#define RD_LE16(a)    (BREW16(*((hcc_u16*)(a))))

/* Read 16 bit big endian value from address. */
#define RD_BE16(a) (*(hcc_u16*)(a))
/* Write 16bit value in v to address a in big endian order. */
#define WR_BE16(a, v) (*(hcc_u16*)(a) = (hcc_u16)(v))
/* Read 32 bit little endian value from address. */
#define RD_BE32(a) (*(hcc_u32*)(a))
/* Write 32bit value in v to address a in big endian order. */
#define WR_BE32(a, v) (*(hcc_u32*)(a) = (hcc_u32)(v))

#endif /*_CMX_TYPES_H_*/

/****************************** END OF FILE **********************************/
```

utils.h:

```
/***************************************************************************
 *
 *            Copyright (c) 2007 by CMX Systems, Inc.
 *
 * This software is copyrighted by and is the sole property of
 * CMX.  All rights, title, ownership, or other interests
 * in the software remain the property of CMX.  This
 * software may only be used in accordance with the corresponding
 * license agreement.  Any unauthorized use, duplication, transmission,
 * distribution, or disclosure of this software is expressly forbidden.
 *
 * This Copyright notice may not be removed or modified without prior
 * written consent of CMX.
 *
 * CMX reserves the right to modify this software without notice.
 *
 * CMX Systems, Inc.
 * 12276 San Jose Blvd. #511
 * Jacksonville, FL 32223
 * USA
 *
 * Tel:  (904) 880-1840
 * Fax:  (904) 880-1632
 * http: www.cmx.com
 * email: cmx@cmx.com
 *
 ******************************************************************************/
#ifndef _UTILS_H_
#define _UTILS_H_

#include "hcc_types.h"

extern void itoa(int number, char* buf, int length);
extern void itoah(int number, char* buf, int length);
extern hcc_u32 strtoi (char *str);
extern void *_memcpy(void *dst, const void *src, int n);
extern void *_memset(void *s, int c, int n);
#endif
/******************************* END OF FILE **********************************/
```

SCIfunctions.h:

```
#ifndef _SCI_FUNCTIONS_H
#define _SCI_FUNCTIONS_H

#include "MC9S08SH8.h"

/* added for terminal */
int TERMIO_kbhit(void);

/*void TERMIO_PutChar(char send);*/
/*modified to be compatible with terminal.c */
int TERMIO_PutChar(char send); /*changed from char*/

int TERMIO_GetChar(void);  /*changed from char*/

void TERMIO_Init_20M_115200(void);

void TERMIO_Init_16M_115200(void);

/*added to initiali8ze clock*/
void ICS_FEI_16M(void);

#endif
```

hcc_terminal.h:

```
/******************************************************************************
*
*            Copyright (c) 2006-2007 by CMX Systems, Inc.
*
* This software is copyrighted by and is the sole property of
* CMX.  All rights, title, ownership, or other interests
* in the software remain the property of CMX.  This
* software may only be used in accordance with the corresponding
* license agreement.  Any unauthorized use, duplication, transmission,
* distribution, or disclosure of this software is expressly forbidden.
*
* This Copyright notice may not be removed or modified without prior
* written consent of CMX.
*
* CMX reserves the right to modify this software without notice.
*
* CMX Systems, Inc.
* 12276 San Jose Blvd. #511
* Jacksonville, FL 32223
* USA
*
* Tel:  (904) 880-1840
* Fax:  (904) 880-1632
* http: www.cmx.com
* email: cmx@cmx.com
*
****************************************************************************/

#ifndef _TERMINAL_H_
#define _TERMINAL_H_
#ifdef __cplusplus
extern "C" {
#endif

typedef void (cmd_func)(char *params);

typedef struct {
  const char *txt;
  cmd_func * func;
  const char *help_txt;
} command_t;

extern int terminal_add_cmd(command_t *cmd);
extern int terminal_delete_cmd(command_t *cmd);
extern void terminal_init(int (*putch)(char), int (*getch)(void), int(*kbhit)(void));
extern void terminal_process(void);
extern int skipp_space(char *cmd_line, int start);
extern int find_word(char *cmd_line, int start);
extern int cmp_str(char *a, char *b);
extern void print(char *s);

#ifdef __cplusplus
}
#endif

#endif

/****************************** END OF FILE **********************************/
```

C Code files:

utils.c:

```
/****************************************************************************
 *
 *            Copyright (c) 2007 by CMX Systems, Inc.
 *
 * This software is copyrighted by and is the sole property of
 * CMX.  All rights, title, ownership, or other interests
 * in the software remain the property of CMX.  This
 * software may only be used in accordance with the corresponding
 * license agreement.  Any unauthorized use, duplication, transmission,
 * distribution, or disclosure of this software is expressly forbidden.
 *
 * This Copyright notice may not be removed or modified without prior
 * written consent of CMX.
 *
 * CMX reserves the right to modify this software without notice.
 *
 * CMX Systems, Inc.
 * 12276 San Jose Blvd. #511
 * Jacksonville, FL 32223
 * USA
 *
 * Tel:  (904) 880-1840
 * Fax:  (904) 880-1632
 * http: www.cmx.com
 * email: cmx@cmx.com
 *
 *****************************************************************************/
#include "utils.h"

void itoah(int number, char* buf, int length)
{
  int ndx=length-1;
  int neg;
  buf[ndx--]='\0';

  if (number < 0)
  {
    neg=1;
    number*=-1;
  }
  else
  {
    neg=0;
  }

  while((number >= 0) && (ndx > 0))
  {
    int digit=number&0xf;
    buf[ndx--]=(char)('0'+digit);
    number>>=4;
  }

  while(ndx > 0)
  {
    buf[ndx--]=' ';
  }

  buf[0]=(char)(neg ? '-' : ' ');
}

void itoa(int number, char* buf, int length)
{
  int ndx=length-1;
  int neg;
  buf[ndx--]='\0';

  if (number < 0)
  {
    neg=1;
    number*=-1;
  }
  else
  {
    neg=0;
  }

  while((number >= 0) && (ndx > 0))
  {
    int digit=number%10;
    buf[ndx--]=(char)('0'+digit);
    number/=10;
  }

  while(ndx > 0)
  {
    buf[ndx--]=' ';
  }
  buf[0]=(char)(neg ? '-' : ' ');
}


hcc_u32 strtoi (char *str)
{
       hcc_u32 rvalue;
  char *c;

       rvalue = 0;

       /* Check for invalid chars in str */
       for ( c = str; *c != '\0'; ++c)
       {
         /* Convert char to num in 0..36 */
         hcc_u8 val=(hcc_u8)(*c-'0');
      rvalue = (rvalue * 10) + val;
       }

  return rvalue;
}

void *_memcpy(void *dest, const void *src, int n)
{
  int x;
  for(x=0; x<n; x++)
    {
      ((char*)dest)[x]=((char*)src)[x];
    }
    return(dest);
}

void *_memset(void *s, int c, int n)
{
  int x;
  for(x=0;x<n;x++)
  {
    ((unsigned char*)s)[x]=(unsigned char)c;
  }
  return(s);
}

/****************************** END OF FILE **********************************/
```

SCI_Functions.c:

```
#include "SCI_Functions.h"


/* function to see if an incoming character is waiting*/
/* normally this would be in SCI_Functions.c/h but it was missing*/
int TERMIO_kbhit(void){
  int i;
  i= SCIS1_RDRF;
  return i; /*returns 0 if no data,1 if data waiting */
}

/* Internal clock source (ICS) initialization */
/* this is from ICS_Functions.c in the SH8App project*/
/**
 * ICS_FEI_16M: This function sets ICS FEI mode and
   16M BUS clock with trim.
 *
 * Parameters:    void
 *
 * Subfunctions:  none.
 *
 * Return:        void
 */
void ICS_FEI_16M(void)
{
  ICSC1_CLKS  = 0;
  ICSC1_IREFS = 1;
  ICSC1_RDIV  = 0;
  ICSC2_BDIV  = 0;
  ICSTRM      = 0x96;
}


/**
 * SPI_PutChar:   This function sends a character through the SPI.
 *
 * Parameters:    character to be sent
 *
 * Subfunctions:  none.
 *
 * Return:        void
 */
/*void TERMIO_PutChar(char send)*/
int TERMIO_PutChar(char send){
  int dummy;
  while(!SCIS1_TDRE);/* wait until TDRE=1 */
  dummy = SCIS1;
  SCID  = send;
  dummy=send;
  return dummy; /*added line*/
}//end SPI_PutChar

/**
 * SPI_GetChar:   This function receives a character through the SPI
 *
 * Parameters:    none
 *
 * Subfunctions:  none.
 *
 * Return:        character recieved
 */

int TERMIO_GetChar(void)
{
  char dummy;
  while(!SCIS1_RDRF);/*wait for RDRF=1 */
  dummy = SCIS1;
  dummy = SCID;
  return dummy;
} //end SPI_GetChar

/**
 * TERMIO_Init_20M_115200: This function initial the terminal.
 *
 * Parameters:             void
 *
 * Subfunctions:           none.
 *
 * Return:                 void
 */
void TERMIO_Init_20M_115200(void)
{
  SCIBDH = 0;
```

```c
  SCIBDL = 0xB;
  SCIC1  = 0;
  SCIC2  = 0x0C;  /*turns on transmitter, receiver*/
}//end TERMIO_Init_20M_115200

/**
 * TERMIO_Init_16M_115200: This function initial the terminal.
 *
 * Parameters:             void
 *
 * Subfunctions:           none.
 *
 * Return:                 void
 */
void TERMIO_Init_16M_115200(void)
{
  SCIBDH = 0;
  SCIBDL = 0x9;
  SCIC1  = 0;
  SCIC2  = 0x0C;  /*turns on transmitter, receiver*/
}//end TERMIO_Init_16M_115200
```

hcc_terminal.c: (this is the big one)

```c
/****************************************************************************
 *
 *            Copyright (c) 2006-2007 by CMX Systems, Inc.
 *
 * This software is copyrighted by and is the sole property of
 * CMX.  All rights, title, ownership, or other interests
 * in the software remain the property of CMX.  This
 * software may only be used in accordance with the corresponding
 * license agreement.  Any unauthorized use, duplication, transmission,
 * distribution, or disclosure of this software is expressly forbidden.
 *
 * This Copyright notice may not be removed or modified without prior
 * written consent of CMX.
 *
 * CMX reserves the right to modify this software without notice.
 *
 * CMX Systems, Inc.
 * 12276 San Jose Blvd. #511
 * Jacksonville, FL 32223
 * USA
 *
 * Tel:  (904) 880-1840
 * Fax:  (904) 880-1632
 * http: www.cmx.com
 * email: cmx@cmx.com
 *
 ***************************************************************************/
#include "hcc_types.h"
#include "hcc_terminal.h"
#ifdef __cplusplus
extern "C" {
#endif

/*****************************************************************************
 * Macro definitions
 *****************************************************************************/
#define MAX_CMDS    10
/*****************************************************************************
 * Local types.
 *****************************************************************************/
/* none */
```

```
/*******************************************************************************
 * External references.
 *******************************************************************************/
/* none */

/*******************************************************************************
 * Function predefinitions.
 *******************************************************************************/
static void print_greeting(void);
static void cmd_help(char *param);
static void print_prompt(void);
static int find_command(char *name);

/*******************************************************************************
 * Module variables.
 *******************************************************************************/
static const command_t help_cmd = {
  "help", cmd_help, "Prints help about commands. "
};

static char cmd_line[0x20];
static hcc_u8 cmd_line_ndx;

static hcc_u8 n_cmd;
static command_t * cmds[MAX_CMDS];

static int (*putch)(char);
static int (*getch)(void);
static int (*kbhit)(void);
/*******************************************************************************
 * Name:
 *    print
 * In:
 *    s: string
 * Out:
 *    n/a
 *
 * Description:
 *    Print the specified string.
 * Assumptions:
 *
 *******************************************************************************/
void print(char *s)
{
char c; /*added*/
  while(*s)
  {
    /*while(*s != (char)putch(*s))*/
    /* Replace with just a CALL TO PUTCH */
    c=(char)putch(*s)
      ;
    s++;
  }
}

/*******************************************************************************
 * Name:
 *    skipp_space
 * In:
 *    cmd_line: string to parse
 *    start: start at offset
 * Out:
 *    index of first non space character
 *
 * Description:
 *
 * Assumptions:
 *
 ******************************************************************************/
int skipp_space(char *cmd_line, int start)
{
  /* Skip leading whitespace. */
  while(cmd_line[start] == ' ' || cmd_line[start] == '\t')
  {
    start++;
  }
  return(start);
}

/******************************************************************************
 * Name:
 *    find_word
 * In:
 *    cmd_line - pointer to string to be processed
 *    start    - start offset of word
 *
 * Out:
 *    Index of end of word.
 *
 * Description:
 *    Will find the end of a word (first space, tab or end of line).
 *
 * Assumptions:
 *    --
 ******************************************************************************/
int find_word(char *cmd_line, int start)
{
  /* Find end of this word. */
  while(cmd_line[start] != ' ' && cmd_line[start] != '\t'
        && cmd_line[start] != '\n' && cmd_line[start] != '\0')
  {
    start++;
  }

  return(start);
}

/******************************************************************************
 * Name:
 *    cmp_str
 * In:
 *    a - pointer to string one
 *    b - pointer to string two
 * Out:
 *    0 - strings differ
 *    1 - strings are the same
 * Description:
 *    Compare two strings.
 *
 * Assumptions:
 *    --
 ******************************************************************************/
int cmp_str(char *a, char *b)
{
  int x=0;
  do
  {
    if (a[x] != b[x])
    {
      return(0);
    }
    x++;
  } while(a[x] != '\0' && b[x] != '\0');

  return(a[x]==b[x] ? 1 : 0);
}

/******************************************************************************
* Name:
*    cmd_help
* In:
*    param - pointer to string containing parameters
*
* Out:
*    N/A
*
* Description:
*    List supported commands.
*
* Assumptions:
*    --
******************************************************************************/
static void cmd_help(char *param)
{
  int x;

  param++;
  print("I understand the following commands:\r\n");

  for(x=0; x < n_cmd; x++)
  {
    print("  ");
    print((char *)cmds[x]->txt);
    print(":\t");
    print((char *)cmds[x]->help_txt);
    print("\r\n");
  }
  print("\r\n");
}


/******************************************************************************
* Name:
*    print_prompt
* In:
*    N/A
*
* Out:
*    N/A
*
* Description:
*    Prints the prompt string.
*
* Assumptions:
*    --
******************************************************************************/
static void print_prompt(void)
{
  print("\r\n>");
}

/******************************************************************************
* Name:
```

```
*    print_greeting
* In:
*    N/A
*
* Out:
*    N/A
*
* Description:
*    --
*
* Assumptions:
*    --
******************************************************************************/
static void print_greeting(void)
{
  print("This is the simple terminal version 1.0\r\n");
}

/******************************************************************************
* Name:
*    find_command
* In:
*    name - pointer to command name string
*
* Out:
*    number - Index of command in "commands" array.
*    -1     - Command not found.
*
* Description:
*    Find a command by its name.
*
* Assumptions:
*    --
******************************************************************************/
static int find_command(char *name)
{
  int x;
  for(x=0; x < n_cmd; x++)
  { /* If command found, execute it. */
    if (cmp_str(name, (char *) cmds[x]->txt))
    {
      return(x);
    }
  }
  return(-1);
}

/******************************************************************************
* Name:
*    terminal_init
* In:
*    N/A
*
* Out:
*    N/A
*
* Description:
*    Inicialise the terminal. Set local varaibles to a default value, print
*    greeting message and prompt.
*
* Assumptions:
*    --
******************************************************************************/
void terminal_init(int (*putch_)(char), int (*getch_)(void), int(*kbhit_)(void))
{
  cmd_line[sizeof(cmd_line)-1]='\0';
  cmd_line_ndx=0;

  cmds[0]=(void *)&help_cmd;
  n_cmd=1;

  putch=putch_;
  getch=getch_;
  kbhit=kbhit_;

  print_greeting();
  print_prompt();
}

/*****************************************************************************
* Name:
*    terminal_proces
* In:
*    N/A
*
* Out:
*    N/A
*
* Description:
*    Main loop of terminal application. gathers input, and executes commands.
*
* Assumptions:
*    --
*****************************************************************************/
void terminal_process(void)
{
  char c;    /* moved up one line */
  while(c=(*kbhit)())
  {
    c=(char)(*getch)();
    /*while(c!=(char)(*putch)(c));*/
    /* Replace with just a simple call to putch()*/
    c=(char)(*putch)(c);
    if (c=='\r')
    {
      while('\n'!=(char)(*putch)('\n'))
        ;
    }
    /* Execute command if enter is received, or cmd_line is full. */
    if ((c=='\r') || (cmd_line_ndx == sizeof(cmd_line)-2))
    {
      int start=skipp_space(cmd_line, 0);
      int end=find_word(cmd_line, start);
      int x;

      /* Separate command string from parameters, and close
         parameters string. */
      cmd_line[end]=cmd_line[cmd_line_ndx]='\0';

      /* Identify command. */
      x=find_command(cmd_line+start);
      /* Command not found. */
      if (x == -1)
      {
        print("Unknown command!\r\n");
      }
      else
      {
        (*cmds[x]->func)(cmd_line+end+1);
      }
      cmd_line_ndx=0;
      print_prompt();
    }
    else
    { /* Put character to cmd_line. */
      if (c=='\b')
      {
        if (cmd_line_ndx > 0)
        {
          cmd_line[cmd_line_ndx]='\0';
          cmd_line_ndx--;
        }
      }
      else
      {
        cmd_line[cmd_line_ndx++]=c;
      }
    }
  }
}

/******************************************************************************
* Name:
*    terminal_add_cmd
* In:
*    N/A
*
* Out:
*    N/A
*
* Description:
*    Main loop of terminal application. gathers input, and executes commands.
*
* Assumptions:
*    --
******************************************************************************/
int terminal_add_cmd(command_t *cmd)
{
  if (n_cmd >= MAX_CMDS)
  {
    return(1);
  }

  cmds[n_cmd]=cmd;
  n_cmd++;
  return(0);
}

/******************************************************************************
* Name:
*    terminal_delete_cmd
* In:
*    N/A
*
* Out:
*    N/A
*
* Description:
*    Main loop of terminal application. gathers input, and executes commands.
*
* Assumptions:
*    --
******************************************************************************/
int terminal_delete_cmd(command_t *cmd)
{
  int x;

  for(x=0; x<n_cmd; x++)
  {
    if (cmds[x] == cmd)
    {
      while(x<n_cmd-1)
      {
        cmds[x]=cmds[x+1];
        x++;
      }

      cmds[x]=0;
      n_cmd--;

      return(0);
    }
  }
  return(1);
}

#ifdef __cplusplus
}
#endif
/****************************** END OF FILE **********************************/
```

main.c:

```
#include <hidef.h> /* for EnableInterrupts macro */
#include "derivative.h" /* include peripheral declarations */
#include "hcc_terminal.h"
#include "utils.h"
#include "SCI_Functions.h"

/* command called function to modify LED state */
static void cmd_led(char *param)
{
  static led_on=0;
  param++;
  if (led_on)
  {
    led_on=0;
    PTBD_PTBD7=0;
  }
  else
  {
    led_on=1;
    PTBD_PTBD7=1;
  }

}

/* the command to modify the LED state */
static const command_t led_cmd = {
  "led", cmd_led, "Toggles leds state."
};

/* function to report ADC results */
static void cmd_adc(char *param){
  static channel=0;
  int i;
  char s[4];
  ADCCFG=0x40; /*10 bit mode*/
  for(i=0;i<10&&param[i]!=0;i++){
    if((param[i]=='1')||(param[i]=='2'))break;
  }
  if(param[i]=='1'){
    /* collect pot value */
    APCTL1=0x01;
    ADCSC1=0;
  } else if(param[i]=='2'){
    /* collect photo value */
    APCTL1=0x02;
    ADCSC1=1;
  }
  while(ADCSC1_COCO!=1);
  i=ADCR;
  s[0]=((i*5)>>10);
  s[1]='.' ;
  i=i-(s[0]<<10);
  s[0]=s[0]+'0';
  s[2]=((i*50)>>10)+'0';
  s[3]=0;
  print(s);
}

/* command for ADC status */
static const command_t adc_cmd = {
  "adc", cmd_adc, "Gives A/D result: 1 for pot / 2 for photo"
};

/* Main program */
void main(void) {
  char c;

  EnableInterrupts; /* enable interrupts */
  /* include your code here */
  PTBDD=0x80;
  ICS_FEI_16M();
  TERMIO_Init_16M_115200();
  terminal_init(TERMIO_PutChar, TERMIO_GetChar, TERMIO_kbhit);
  (void)terminal_add_cmd((command_t*)&led_cmd);
  (void)terminal_add_cmd((command_t*)&adc_cmd);
  /* main loop */
  for(;;) {
    c=SCIS1;
    terminal_process(); /* services terminal, dispatches commands*/
    DisableInterrupts;
    __RESET_WATCHDOG(); /* feeds the dog */
    EnableInterrupts;
  } /* loop forever */
  /* please make sure that you never leave main */
}
```