```
/**-------------------------------------------------------------------------
            \file Monitor.cpp
--                                                                        --
--              ECEN 5803 Mastering Embedded Systems Architecture          --
--                  Project 1 Module 3                                    --
--                Microcontroller Firmware                                --
--                      Monitor.cpp                                         --
--                                                                        --
-------------------------------------------------------------------------
--
--  Designed for:  University of Colorado at Boulder
--
--
--  Designed by:  Tim Scherr
--  Revised by:  David James & Ismail Yesildirek
--
-- Version: 2.0.1
-- Date of current revision:  2018-10-03
-- Target Microcontroller: Freescale MKL25ZVMT4
-- Tools used:  ARM mbed compiler
--              ARM mbed SDK
--              Keil uVision MDK v.5
--              Freescale FRDM-KL25Z Freedom Board
--
--
   Functional Description: See below
--
--      Copyright (c) 2015 Tim Scherr All rights reserved.
--
*/

#include <stdio.h>
#include "shared.h"
DigitalOut greenLED(LED_GREEN);
bool green_led_status = 1; //default is on.
/*****************************************************************************
*  \fn uint32_t getR0(void)
* assembly routine which returns the unaltered contents of register r0
*****************************************************************************/
__asm uint32_t getR0()
{
    BX lr ; returns r0 without altering it
}

/*****************************************************************************
*  \fn void getRn(uint32_t[16])
* assembly routine which fills a uint32_t[16] array's 1-15 elements with r1-r15
*****************************************************************************/
__asm void getRn(uint32_t reglist[16])
{
    STR r1, [r0, #4]  ; r1->reglist[1]
    STR r2, [r0, #4]  ; r2->reglist[2]
    STR r3, [r0, #12] ; r3->reglist[3]
    STR r4, [r0, #16] ; r4->reglist[4]
    STR r5, [r0, #20] ; r5->reglist[5]
    STR r6, [r0, #24] ; r6->reglist[6]
    STR r7, [r0, #28] ; r7->reglist[7]
    MOV r1, r8        ; STR only takes r0-r7
    STR r1, [r0, #32] ; r8->r1->reglist[8]
    MOV r1, r9        ; STR only takes r0-r7
    STR r1, [r0, #36] ; r9->r1->reglist[9]
    MOV r1, r10       ; STR only takes r0-r7
    STR r1, [r0, #40] ; r10->r1->reglist[10]
    MOV r1, r11       ; STR only takes r0-r7
    STR r1, [r0, #44] ; r11->r1->reglist[11]
    MOV r1, r12       ; STR only takes r0-r7
    STR r1, [r0, #48] ; r12->r1->reglist[12]
    MOV r1, sp        ; STR only takes r0-r7
    STR r1, [r0, #52] ; sp(r13)->r1->reglist[13]
    MOV r1, lr        ; STR only takes r0-r7
    STR r1, [r0, #56] ; lr(r14)->r1->reglist[14]
```

```
72        MOV r1, pc        ; STR only takes r0-r7
73        STR r1, [r0, #60] ; pc(r15)->r1->reglist[15]
74        BX lr             ; return
75    }
76
77    /**************************************************************************
78    *  \fn uint32_t getWord(uint32_t) assembly routine
79    * returns the 32 bit word stored at the 32 bit address in the argument
80    **************************************************************************/
81    __asm uint32_t getWord(uint32_t address)
82    {
83      LDR r0, [r0]  ; data @ r0 -> r0
84      BX lr         ; return
85    }
86
87    /**************************************************************************
88    *  \fn void show_regs_and_mem(void)
89    * prints registers r0-r15 to debug port,
90    * and shows the contents of 16 words worth of memory.
91    * The memory addresses cycle from 0x0 to 0x6000
92    **************************************************************************/
93    void show_regs_and_mem()
94    {
95      // The top line of the display
96      UART_direct_msg_put("\r\n\r\nRegister contents:");
97      UART_direct_msg_put("\t|\tADDRESS:\tDATA:");
98
99      // retrieve the contents of registers r0-r15
100     uint32_t reg[16]; // array to hold r0-r15
101     reg[0] = getR0(); // because passing any variable would alter r0
102     getRn(reg);       // fill the rest of the array
103     uint8_t i=0;      // 8-bit unsigned index variable
104
105     // addr is declared static so we can cycle through addresses
106     static uint32_t addr = 0x0; // start at 0x00000000
107     uint32_t data;        // a variable to hold the data
108     // print 16 lines:
109     for(i=0;i<16;i++)
110     {
111       UART_direct_msg_put("\r\nr"); //start a new line with r
112       //now, to print the register number:
113       if(i/10)  // if the index is greater than 10...
114       { // convert the hex value to decimal
115         UART_direct_hex_put((i/10 << 4) + i%10);
116       }
117       else
118       { // otherwise just print the bottom nibble
119         UART_low_nibble_direct_put(i&0xF);
120       }                                  //add tab for alignment,
121       UART_direct_msg_put(" \t0x");      // format hex values with 0x########
122       UART_direct_word_hex_put(reg[i]); //print the whole word
123       UART_direct_msg_put("\t|\t0x");    // tabs and a divider, plus 0x########
124       UART_direct_word_hex_put(addr);    // first print the address
125       UART_direct_msg_put("\t0x");       // tab and add 0x########
126       data = getWord(addr);              // get the data @ addr
127       UART_direct_word_hex_put(data);    // print the data
128       if(addr < 0x6000) addr +=4;        // increment the address by 4
129       else addr = 0;                     // roll over at 0x6000
130     }
131     return;
132   }
133   /**************************************************************************
134   * \fn hex2hexInt converts a hex value to a hex representation of its decimal value
135   *
136   **************************************************************************/
137   uint32_t hex2hexInt(uint32_t hex)
138   {
139     if(hex > 99999999) return hex;
140     uint32_t hexInt = 0;
141     uint32_t tempHex = hex;
142     uint8_t i = 0;
```

```
143     for(i=0;i<8;i++)
144     {
145     hexInt += (tempHex%10)<<(i*4);
146     tempHex = tempHex/10;
147     }
148     return hexInt;
149   }
150
151
152
153   /*************************************************************************
154   * Set Display Mode Function
155   * Function determines the correct display mode.  The 3 display modes operate as
156   *   follows:
157   *
158   *  NORMAL MODE       Outputs only mode and state information changes
159   *                     and calculated outputs
160   *
161   *  QUIET MODE        No Outputs
162   *
163   *  DEBUG MODE        Outputs mode and state information, error counts,
164   *                    register displays, sensor states, and calculated output
165   *
166   *
167   * There is deliberate delay in switching between modes to allow the RS-232 cable
168   * to be plugged into the header without causing problems.
169   *************************************************************************/
170
171   void set_display_mode(void)
172   {
173     UART_direct_msg_put("\r\nSelect Mode");
174     UART_direct_msg_put("\r\n Hit NOR - Normal");
175     UART_direct_msg_put("\r\n Hit QUI - Quiet");
176     UART_direct_msg_put("\r\n Hit DEB - Debug" );
177     UART_direct_msg_put("\r\n Hit V - Version#");
178     UART_direct_msg_put("\r\n Hit L - Toggle Green LED");
179     UART_direct_msg_put("\r\nSelect:  ");
180   }
181
182   /*************************************************************************
183   * \fn void chk_UART_msg(void)
184   *
185   *************************************************************************/
186   void chk_UART_msg(void)
187   {
188      UCHAR j;
189      while( UART_input() )       // becomes true only when a byte has been received
190      {                                  // skip if no characters pending
191         j = UART_get();                 // get next character
192
193         if( j == '\r' )           // on a enter (return) key press
194         {                 // complete message (all messages end in carriage return)
195            UART_msg_process();
196         }
197         else
198         {
199            if ((j != 0x02) )          // if not ^B
200            {                               // if not command, then
201               UART_direct_put(j);               // echo the character
202            }
203            if( j == '\b' )
204            {                               // backspace editor
205               if( msg_buf_idx != 0)
206               {                          // if not 1st character then destructive
207                  UART_direct_msg_put(" \b");// backspace
208                  msg_buf_idx--;
209               }
210            }
211            else if( msg_buf_idx >= MSG_BUF_SIZE )
212            {                                  // check message length too large
213               msg_buf_idx = 0;
```

```
214            }
215            else if ((display_mode == QUIET) && (msg_buf[0] != 0x02) &&
216                     (msg_buf[0] != 'D') && (msg_buf[0] != 'd') &&
217                     (msg_buf[0] != 'N') && (msg_buf[0] != 'n') &&
218                     (msg_buf[0] != 'V') && (msg_buf[0] != 'v') &&
219                     (msg_buf[0] != 'L') && (msg_buf[0] != 'l') &&
220                     (msg_buf_idx != 0))
221            {                           // if first character is bad in Quiet mode
222               msg_buf_idx = 0;         // then start over
223            }
224            else {                         // not complete message, store character
225
226               msg_buf[msg_buf_idx] = j;
227               msg_buf_idx++;
228            }
229         }
230      }
231   }
232
233   /*************************************************************************
234   *  \fn void UART_msg_process(void)
235   *UART Input Message Processing
236   *************************************************************************/
237   void UART_msg_process(void)
238   {
239      UCHAR chr,err=0;
240      chr = msg_buf[0];
241   //   unsigned char  data;
242
243         switch( chr )
244         {
245            case 'D':
246            case 'd':
247               if((msg_buf[1] == 'E' || msg_buf[1] == 'e') &&
248                  (msg_buf[2] == 'B' || msg_buf[2] == 'b'))
249               {
250                  display_mode = DEBUG;
251                  UART_direct_msg_put("\r\nMode=DEBUG\n");
252                  display_timer = 0;
253               }
254               else
255                  err = 1;
256               break;
257
258            case 'N':
259            case 'n':
260               if((msg_buf[1] == 'O' || msg_buf[1] == 'o') &&
261                  (msg_buf[2] == 'R' || msg_buf[2] == 'r'))
262               {
263                  display_mode = NORMAL;
264                  UART_direct_msg_put("\r\nMode=NORMAL\n");
265                  //display_timer = 0;
266               }
267               else
268                  err = 1;
269               break;
270
271            case 'Q':
272            case 'q':
273               if((msg_buf[1] == 'U' || msg_buf[1] == 'u') &&
274                  (msg_buf[2] == 'I' || msg_buf[2] == 'i'))
275               {
276                  display_mode = QUIET;
277                  UART_direct_msg_put("\r\nMode=QUIET\n");
278                  display_timer = 0;
279               }
280               else
281                  err = 1;
282               break;
283
284            case 'V':
```

```
285            case 'v':
286               //display_mode = VERSION;
287               UART_direct_msg_put("\r\n");
288               UART_direct_msg_put( CODE_VERSION );
289               display_timer = 0;
290               break;
291
292            case 'L':
293            case 'l':
294               greenLED = !greenLED;
295               green_led_status = !green_led_status;
296               //display_mode = LED;
297               UART_direct_msg_put("\r\nGreen LED");
298               if (green_led_status ==0){
299               UART_direct_msg_put(" OFF");
300               }
301               else
302               {
303               UART_direct_msg_put(" ON");
304               }
305               display_timer = 0;
306               break;
307
308            default:
309             err = 1;
310      }
311
312      if( err == 1 )
313      {
314         UART_direct_msg_put("\n\rError!");
315      }
316      msg_buf_idx = 0;          // put index to start of buffer for next message
317   }
318
319
320   /*****************************************************************************
321   *   \fn   is_hex
322   * Function takes
323   *  @param a single ASCII character and returns
324   *  @return 1 if hex digit, 0 otherwise.
325   *
326   *****************************************************************************/
327   /*
328   UCHAR is_hex(UCHAR c)
329   {
330      if( (((c |= 0x20) >= '0') && (c <= '9')) || ((c >= 'a') && (c <= 'f'))  )
331         return 1;
332      return 0;
333   }
334   */
335   /*****************************************************************************
336   *   \fn  output flow, temperature and velocity
337   *****************************************************************************/
338   void status_report()
339   {
340     if(display_mode == DEBUG)
341     {//decimal printouts are commented out for debug
342
343     //UART_direct_msg_put("\r\nFlow (GPM): ");
344     //UART_direct_hex_int_put(hex2hexInt(Flow), 2);
345     UART_direct_msg_put("\r\nFlow (GPM): 0x");
346     UART_direct_word_hex_put(Flow);
347     //UART_direct_msg_put("\r\nTemp (C): ");
348     //UART_direct_hex_int_put(hex2hexInt(temperature), 2);
349     UART_direct_msg_put("\r\nTemp (C): 0x");
350     UART_direct_word_hex_put(temperature);
351     //UART_direct_msg_put("\r\nFreq (Hz): ");
352     //UART_direct_hex_int_put(hex2hexInt(frequency), 2);
353     UART_direct_msg_put("\r\nFreq (Hz): 0x");
354     UART_direct_word_hex_put(frequency);
355
```

```
356     // These variables need to be made available in shared.h and main.cpp
357       // if you want to print them.
358     //UART_direct_msg_put("\r\nVelocity: ");
359     //UART_direct_hex_int_put(hex2hexInt(velocity), 2);
360     //UART_direct_msg_put("\r\nVelocity: ");
361     //UART_direct_word_hex_put(velocity);
362     //UART_direct_msg_put("\r\nViscosity (x10^-6): ");
363     //UART_direct_hex_int_put(hex2hexInt(viscosity), 0);
364     //UART_direct_msg_put("\r\nDensity: ");
365     //UART_direct_hex_int_put(hex2hexInt(rho_density),0);
366     //UART_direct_msg_put("\r\nSt: ");
367     //UART_direct_hex_int_put(hex2hexInt(St_const),0);
368     //UART_direct_msg_put("\r\nRe: ");
369     //UART_direct_hex_int_put(hex2hexInt(Re),0);
370     }
371     else
372     {
373       UART_direct_msg_put("\r\nFlow (GPM): ");
374       UART_direct_hex_int_put(hex2hexInt(Flow), 2);
375       UART_direct_msg_put("\r\nTemp (C): ");
376       UART_direct_hex_int_put(hex2hexInt(temperature), 2);
377       UART_direct_msg_put("\r\nFreq (Hz): ");
378       UART_direct_hex_int_put(hex2hexInt(frequency), 2);
379     }
380   }
381   /*****************************************************************************
382   *   \fn  DEBUG and DIAGNOSTIC Mode UART Operation
383   *****************************************************************************/
384   void monitor(void)
385   {
386
387   /*********************************/
388   /*     Spew outputs              */
389   /*********************************/
390
391      switch(display_mode)
392      {
393         case(QUIET):
394            {
395                display_flag = 0;
396            }
397            break;
398
399         case(NORMAL):
400            {
401               if (display_flag == 1)
402               {
403                  UART_direct_msg_put("\r\n\r\nNORMAL ");
404                  status_report();
405                  display_flag = 0;
406               }
407            }
408            break;
409
410         case(DEBUG):
411            {
412               if (display_flag == 1)
413               {
414                  UART_direct_msg_put("\r\n\r\nDEBUG ");
415                  //show_regs_and_mem(); // function displays register contents over UART
416                  status_report();
417                  //  Create a command to read 16 words from the current stack
418                  // and display it in reverse chronological order.
419
420                  display_flag = 0;
421                }
422            }
423            break;
424
425         default:
426         {
```

```
427             UART_direct_msg_put("Mode Error");
428           }
429       }
430   }
431
432
```