

```

1  /**-----
2      \file UART_poll.cpp
3
4  --
5      ECEN 5803 Mastering Embedded System Architecture
6      Project 1 Module 3
7      Microcontroller Firmware
8      UART_poll.c
9  --
10 -----
11 --
12 -- Designed for: University of Colorado at Boulder
13 --
14 --
15 -- Designed by: Tim Scherr
16 -- Revised by: David James & Ismail Yesildirek
17 --
18 -- Version: 2.0.1
19 -- Date of current revision: 2018-10-03
20 -- Target Microcontroller: Freescale MKL25ZVMT4
21 -- Tools used: ARM mbed compiler
22 --               ARM mbed SDK
23 --               Keil uVision MDK v.5
24 --               Freescale FRDM-KL25Z Freedom Board
25 --
26 --
27 -- Functional Description: This file contains routines that support messages
28 -- to and from the UART port. Included are:
29 --     Serial() - a routine to send/receive bytes on the UART port to
30 --                 the transmit/receive buffers
31 --     UART_put() - a routine that puts a character in the transmit buffer
32 --     UART_get() - a routine that gets the next character from the receive
33 --                 buffer
34 --     UART_msg_put() - a routine that puts a string in the transmit buffer
35 --     UART_direct_msg_put() - routine that sends a string out the UART port
36 --     UART_input() - determines if a character has been received
37 --     UART_hex_put() - a routine that puts a hex byte in the transmit buffer
38 --     UART_direct_hex_put - a routine that puts a hex byte directly to the UART port
39 --
40 -- NEW TO VERSION 2.0.1:
41 --     UART_low_nibble_direct_put() - puts the low nibble of a byte in hex directly
42 --                                     (no ram buffer) to the UART.
43 --     UART_direct_word_hex_put() - puts a word in hex directly to the UART
44 -- Copyright (c) 2015 Tim Scherr All rights reserved.
45 --
46 */
47
48
49
50 /*****/
51 /* Configurations */
52 /*****/
53 /*
54
55 */
56
57 #include <stdio.h>
58 #include "shared.h"
59 #include "MKL25Z4.h"
60
61 // NOTE: UART0 is also called UARTLP in mbed
62 #define OERR (UART0->S1 & UARTLP_S1_OR_MASK) // Overrun Error bit
63 #define CREM (UART0->C2 & UARTLP_C2_RE_MASK) // continuous receive enable bit
64 #define RCREG UART0->D // Receive Data Register
65 #define FERR (UART0->S1 & UARTLP_S1_FE_MASK) // Framing Error bit
66 #define RCIF (UART0->S1 & UARTLP_S1_RDRF_MASK) // Receive Interrupt Flag (full)
67 #define TXIF (UART0->S1 & UARTLP_S1_TDRE_MASK) // Transmit Interrupt Flag (empty)
68 #define TXREG UART0->D // Transmit Data Register
69 #define TRMT (UART0->S1 & UARTLP_S1_TC_MASK) // Transmit Shift Register Empty
70
71 /*****/

```

```

72      *          Start of code          *
73      *****/
74
75      UCHAR error_count = 0;
76
77      /// \fn void serial(void)
78      /// function polls the serial port for Rx or Tx data
79      void serial(void)          // The serial function polls the serial port for
80                                // received data or data to transmit
81      {
82                                // deals with error handling first
83      if ( OERR )                // if an overrun error, clear it and continue.
84      {
85          error_count++;
86                                // resets and sets continous receive enable bit
87          UART0->C2 = UART0->C2 & (!UARTLP_C2_RE_MASK);
88          UART0->C2 = UART0->C2 | UARTLP_C2_RE_MASK;
89      }
90
91      if ( FERR){                // if a framing error, read bad byte, clear it and continue.
92          error_count++;
93          RCREG;                 // This will also clear RCIF if only one byte has been
94                                // received since the last int, which is our assumption.
95
96                                // resets and sets continous receive enable bit
97          UART0->C2 = UART0->C2 & (!UARTLP_C2_RE_MASK);
98          UART0->C2 = UART0->C2 | UARTLP_C2_RE_MASK;
99      }
100     else                        // else if no frame error,
101     {
102         if ( RCIF )            // Check if we have received a byte
103         {                      // Read byte to enable reception of more bytes
104                                // For PIC, RCIF automatically cleared when RCREG is read
105                                // Also true of Freescale KL25Z
106                                *rx_in_ptr++ = RCREG;          /* get received character */
107                                if( rx_in_ptr >= RX_BUF_SIZE + rx_buf )
108                                {
109                                    rx_in_ptr = rx_buf;        /* if at end of buffer, circles rx_in_ptr
110                                                                    to top of buffer */
111                                }
112         }
113     }
114 }
115
116 if (TXIF)                      // Check if transmit buffer empty
117 {
118     if ((tx_in_ptr != tx_out_ptr) && (display_mode != QUIET))
119     {
120         TXREG = *tx_out_ptr++;    /* send next char */
121         if( tx_out_ptr >= TX_BUF_SIZE + tx_buf )
122             tx_out_ptr = tx_buf;    /* 0 <= tx_out_idx < TX_BUF_SIZE */
123         tx_in_progress = true;      /* flag needed to start up after idle */
124     }
125     else
126     {
127         tx_in_progress = false;    /* no more to send */
128     }
129 }
130 // serial_count++;              // increment serial counter, for debugging only
131 serial_flag = 1;                // and set flag
132 }
133
134 /* The function UART_direct_msg_put puts a null terminated string directly
135 * (no ram buffer) to the UART in ASCII format.
136 *****/
137 void UART_TX_wait()
138 {
139     while( TXIF == 0 );
140 }
141
142

```

```

143 /*****
144 * The function UART_direct_msg_put puts a null terminated string directly
145 * (no ram buffer) to the UART in ASCII format.
146 *****/
147 void UART_direct_msg_put(const char *str)
148 {
149     while( *str != '\0' )
150     {
151         TXREG = *str++;
152         while( TXIF == 0 || TRMT == 0 ); // waits here for UART transmit buffer
153                                         // to be empty
154     }
155 }
156
157 /*****
158 * The function UART_put puts a byte, to the transmit buffer at the location
159 * pointed to by tx_in_idx. The pointer is incremented circularly as described
160 * previously. If the transmit buffer should wrap around (should be designed
161 * not to happen), data will be lost. The serial interrupt must be temporarily
162 * disabled since it reads tx_in_idx and this routine updates tx_in_idx which is
163 * a 16 bit value.
164 *****/
165 /*
166 void UART_put(UCHAR c)
167 {
168     *tx_in_ptr++ = c; // save character to transmit buffer
169     if( tx_in_ptr >= TX_BUF_SIZE + tx_buf )
170         tx_in_ptr = tx_buf; // 0 <= tx_in_idx < TX_BUF_SIZE
171 }
172 */
173 /*****
174 * The function UART_direct_put puts a ncharacter directly
175 * (no ram buffer) to the UART in ASCII format.
176 *****/
177 void UART_direct_put(UCHAR c)
178 {
179     TXREG = c;
180     UART_TX_wait();
181 }
182 /*****
183 * The function UART_get gets the next byte if one is available from the receive
184 * buffer at the location pointed to by rx_out_idx. The pointer is circularly
185 * incremented and the byte is returned in R7. Should no byte be available the
186 * function will wait until one is available. There is no need to disable the
187 * serial interrupt which modifies rx_in_idx since the function is looking for a
188 * compare only between rx_in_idx & rx_out_idx.
189 *****/
190 UCHAR UART_get(void)
191 {
192     UCHAR c;
193     while( rx_in_ptr == rx_out_ptr ); // wait for a received character,
194                                     // indicated */
195                                     // when pointers are different
196                                     // this could be an infinite loop, but
197                                     // is not because of UART_input check
198     c = *rx_out_ptr++;
199     if( rx_out_ptr >= RX_BUF_SIZE + rx_buf ) // if at end of buffer
200     {
201         rx_out_ptr = rx_buf; // 0 <= rx_out_idx < RX_BUF_SIZE */
202                             // return byte from beginning of buffer
203     }
204     return(c);
205 }
206
207 /*****
208 * The function UART_input returns a 1 if 1 or more receive byte(s) is(are)
209 * available and a 0 if the receive buffer rx buf is empty. There is no need to
210 * disable the serial interrupt which modifies rx_in_idx since function is
211 * looking for a compare only between rx_in_idx & rx_out_idx.
212 *****/
213

```

```

214  UCHAR UART_input(void)
215  {
216      if( rx_in_ptr == rx_out_ptr )return(0);
217          /* no characters in receive buffer */
218      return(1);          /* 1 or more receive characters ready */
219  }
220
221  /*****
222  * The function UART_msg_put puts a null terminated string through the transmit
223  * buffer to the UART port in ASCII format.
224  *****/
225  /*
226  void UART_msg_put(const char *str)
227  {
228      while( *str != '\0' )
229      {
230          *tx_in_ptr++ = *str++;          // save character to transmit buffer
231          if( tx_in_ptr >= TX_BUF_SIZE + tx_buf)
232              tx_in_ptr = tx_buf;          // 0 <= tx_in_idx < TX_BUF_SIZE
233      }
234  }
235  */
236  /*****
237  * HEX_TO_ASC Function
238  * Function takes a single hex character (0 thru Fh) and converts to ASCII.
239  *****/
240  UCHAR hex_to_asc(UCHAR c)
241  {
242      if( c <= 9 )
243          return( c + 0x30 );
244      return( ((c & 0x0f) + 0x37) );          /* add 37h */
245  }
246
247  /*****
248  * ASC_TO_HEX Function
249  * Function takes a single ASCII character and converts to hex.
250  *****/
251  /*
252  UCHAR asc_to_hex(UCHAR c)
253  {
254      if( c <= '9' )
255          return( c - 0x30 );
256      return( (c & 0xdf) - 0x37 );          // clear bit 5 (lower case) & subtract 37h
257  }
258  */
259
260  /*****
261  * The function UART_low_nibble_put puts the low nibble of a byte in hex directly
262  * (no ram buffer) to the UART.
263  *****/
264  void UART_low_nibble_direct_put(UCHAR c)
265  {
266      TXREG = hex_to_asc( c & 0x0f );
267      UART_TX_wait();
268  }
269
270  /*****
271  * The function UART_high_nibble_put puts the high nibble of a byte in h
272  * UART port.
273  *****/
274  //void UART_high_nibble_put(unsigned char c)
275  //{
276  //    UART_put( hex_to_asc( (c>>4) & 0x0f ));
277  //}
278
279  /*****
280  * The function UART_hex_put puts 1 byte in hex through the transmit buffer to
281  * the UART port.
282  *****/
283  /*
284  void UART_hex_put(unsigned char c)

```

```

285 {
286     UART_put( hex_to_asc( (c>>4) & 0x0f )); // could eliminate & as >> of UCHAR
287                                             // by definition clears upper bits.
288     UART_put( hex_to_asc( c & 0x0f ));
289 }
290 */
291 /*****
292 * The function UART_direct_hex_put puts 1 byte in hex directly (no ram buffer)
293 * to the UART.
294 *****/
295 void UART_direct_hex_put(unsigned char c)
296 {
297     TXREG = hex_to_asc( (c>>4) & 0x0f );
298     UART_TX_wait();
299     TXREG = hex_to_asc( c & 0x0f );
300     UART_TX_wait();
301 }
302 /*****
303 * The function UART_direct_hex_put puts 4 bytes in hex directly (no ram buffer)
304 * to the UART.
305 *****/
306 void UART_direct_word_hex_put(uint32_t word)
307 {
308     UART_direct_hex_put((word>>24)&0xFF);
309     UART_TX_wait();
310     UART_direct_hex_put((word>>16)&0xFF);
311     UART_TX_wait();
312     UART_direct_hex_put((word>>8)&0xFF);
313     UART_TX_wait();
314     UART_direct_hex_put(word&0xFF);
315     UART_TX_wait();
316 }
317 /*****
318 * The function UART_direct_hex_int_put removes the leading zeroes and adds
319 * a decimal point. Best used in conjunction with hex2hexInt
320 *****/
321 void UART_direct_hex_int_put(uint32_t word, uint8_t deci)
322 {
323     bool zeros = true;
324     int8_t i = 7; //must be signed because of wrap-around
325     for(i=7;i>=0;i--)
326     {
327         if(zeros && ((word>>(i*4))&0xF)==0);
328         else
329         {
330             zeros = false;
331             UART_low_nibble_direct_put((word>>i*4)&0xF);
332             if(i == deci & deci>0) UART_direct_put('.');
333         }
334     }
335 }
336

```