

```

1 /* SERIAL.C: This code is designed to act as a low-level serial driver for
2    higher-level programming. Ideally, one could simply call init_serial()
3    to initialize the serial port, then use serial_send("data", 4) to send
4    an array of data (8-bit unsigned character strings).
5
6    WRITTEN BY : Paul Miller <pamiller@uiuc.edu>
7    $Id: serial.c,v 1.4 2003/08/08 16:08:56 paul Exp $
8 */
9 #include <stdio.h>
10 #include <stdlib.h>
11 #include <stdarg.h>
12 #include <string.h>
13 #include <math.h>
14 #include <limits.h>
15
16 #include "F28x_Project.h"    // Device Headerfile and Examples Include File
17 #include <buffer.h>
18 #include <F28379dSerial.h>
19 #include <F2837xD_sci.h>
20
21 serialSCIA_t SerialA;
22 serialSCIB_t SerialB;
23 serialSCIC_t SerialC;
24 serialSCID_t SerialD;
25
26 char RXAdata = 0;
27 char RXBdata = 0;
28 char RxCdata = 0;
29 char RxDdata = 0;
30 uint32_t numRXA = 0;
31 uint32_t numRXB = 0;
32 uint32_t numRXC = 0;
33 uint32_t numRXD = 0;
34
35
36 // for SerialA
37 uint16_t init_serialSCIA(serialSCIA_t *s, uint32_t baud)
38 {
39     volatile struct SCI_REGS *sci;
40     uint32_t clk;
41
42     if (s == &SerialA) {
43         sci = &SciaRegs;
44         s->sci = sci;
45         init_bufferSCIA(&s->TX);
46
47         GPIO_SetupPinMux(43, GPIO_MUX_CPU1, 15);
48         GPIO_SetupPinOptions(43, GPIO_INPUT, GPIO_PULLUP);
49         GPIO_SetupPinMux(42, GPIO_MUX_CPU1, 15);
50         GPIO_SetupPinOptions(42, GPIO_OUTPUT, GPIO_PUSHPULL);
51
52     } else {
53         return 1;
54     }
55
56     /* init for standard baud,8N1 comm */
57     sci->SCICTL1.bit.SWRESET = 0;    // init SCI state machines and opt flags

```

```

58     sci->SCICCR.all = 0x0;
59     sci->SCICTL1.all = 0x0;
60     sci->SCICTL2.all = 0x0;
61     sci->SCIPRI.all = 0x0;
62     clk = LSPCLK_HZ;                // set baud rate
63     clk /= baud*8;
64     clk--;
65     sci->SCILBAUD.all = clk & 0xFF;
66     sci->SCIHBAUD.all = (clk >> 8) & 0xFF;
67
68     sci->SCICCR.bit.SCICHAR = 0x7;    // (8) 8 bits per character
69     sci->SCICCR.bit.PARITYENA = 0;    // (N) disable party calculation
70     sci->SCICCR.bit.STOPBITS = 0;    // (1) transmit 1 stop bit
71     sci->SCICCR.bit.LOOPBKENA = 0;    // disable loopback test
72     sci->SCICCR.bit.ADDRIDLE_MODE = 0; // idle-line mode (non-multiprocessor SCI comm)
73
74     sci->SCIFFCT.bit.FFTXDLY = 0;    // TX: zero-delay
75
76     sci->SCIFFTX.bit.SCIFFENA = 1;    // enable SCI fifo enhancements
77     sci->SCIFFTX.bit.TXFIFORESET = 0;
78     sci->SCIFFTX.bit.TXFFIL = 0x0; // TX: fifo interrupt at all levels      ??? is this correct
79     sci->SCIFFTX.bit.TXFFINTCLR = 1; // TX: clear interrupt flag
80     sci->SCIFFTX.bit.TXFFIENA = 0;    // TX: disable fifo interrupt
81     sci->SCIFFTX.bit.TXFIFORESET = 1;
82
83     sci->SCIFFRX.bit.RXFIFORESET = 0; // RX: fifo reset
84     sci->SCIFFRX.bit.RXFFINTCLR = 1;  // RX: clear interrupt flag
85     sci->SCIFFRX.bit.RXFFIENA = 1;    // RX: enable fifo interrupt
86     sci->SCIFFRX.bit.RXFFIL = 0x1;    // RX: fifo interrupt
87     sci->SCIFFRX.bit.RXFIFORESET = 1; // RX: re-enable fifo
88
89     sci->SCICTL2.bit.RXBKINTENA = 0;   // disable receiver/error interrupt
90     sci->SCICTL2.bit.TXINTENA = 0;    // disable transmitter interrupt
91
92     sci->SCICTL1.bit.TXWAKE = 0;
93     sci->SCICTL1.bit.SLEEP = 0;        // disable sleep mode
94     sci->SCICTL1.bit.RXENA = 1;        // enable SCI receiver
95     sci->SCICTL1.bit.RXERRINTENA = 0; // disable receive error interrupt
96     sci->SCICTL1.bit.TXENA = 1;        // enable SCI transmitter
97     sci->SCICTL1.bit.SWRESET = 1;      // re-enable SCI
98
99     /* enable PIE interrupts */
100    if (s == &SerialA) {
101        PieCtrlRegs.PIEIER9.bit.INTx1 = 1;
102        PieCtrlRegs.PIEIER9.bit.INTx2 = 1;
103        IER |= (M_INT9);
104        PieCtrlRegs.PIEACK.all = (PIEACK_GROUP9);
105    }
106
107    return 0;
108 }
109
110 void uninit_serialSCIA(serialSCIA_t *s)
111 {
112     volatile struct SCI_REGS *sci = s->sci;
113
114

```

```

115  /* disable PIE interrupts */
116  if (s == &SerialA) {
117      PieCtrlRegs.PIEIER9.bit.INTx1 = 0;
118      PieCtrlRegs.PIEIER9.bit.INTx2 = 0;
119      IER &= ~M_INT9;
120  }
121  sci->SCICTL1.bit.RXERRINTENA = 0;    // disable receive error interrupt
122  sci->SCICTL2.bit.RXBKINTENA = 0;    // disable receiver/error interrupt
123  sci->SCICTL2.bit.TXINTENA = 0;      // disable transmitter interrupt
124
125  sci->SCICTL1.bit.RXENA = 0;          // disable SCI receiver
126  sci->SCICTL1.bit.TXENA = 0;          // disable SCI transmitter
127 }
128
129
130
131 /*****
132  * SERIAL_SEND()
133  *
134  * "User level" function to send data via serial. Return value is the
135  * length of data successfully copied to the TX buffer.
136  *****/
137 uint16_t serial_sendSCIA(serialSCIA_t *s, char *data, uint16_t len)
138 {
139     uint16_t i = 0;
140     if (len && s->TX.size < BUF_SIZE_SCI_A) {
141         for (i = 0; i < len; i++) {
142             if (buf_writeSCIA_1(&s->TX, data[i] & 0x00FF) != 0) break;
143         }
144         s->sci->SCIFFTX.bit.TXFFINTCLR = 1; // TX: clear interrupt flag
145         s->sci->SCIFFTX.bit.TXFFIENA = 1;   // TX: enable fifo interrupt
146     }
147     return i;
148 }
149
150 // For SerialB
151 uint16_t init_serialSCIB(serialSCIB_t *s, uint32_t baud)
152 {
153     volatile struct SCI_REGS *sci;
154     uint32_t clk;
155
156     if (s == &SerialB) {
157         sci = &ScibRegs;
158         s->sci = sci;
159         init_bufferSCIB(&s->TX);
160
161         GPIO_SetupPinMux(15, GPIO_MUX_CPU1, 2);
162         GPIO_SetupPinOptions(15, GPIO_INPUT, GPIO_PULLUP);
163         GPIO_SetupPinMux(14, GPIO_MUX_CPU1, 2);
164         GPIO_SetupPinOptions(14, GPIO_OUTPUT, GPIO_PUSHPULL);
165
166     } else {
167         return 1;
168     }
169 }
170
171 /* init for standard baud, 8N1 comm */

```

```

172     sci->SCICTL1.bit.SWRESET = 0;          // init SCI state machines and opt flags
173     sci->SCICCR.all = 0x0;
174     sci->SCICTL1.all = 0x0;
175     sci->SCICTL2.all = 0x0;
176     sci->SCIPRI.all = 0x0;
177     clk = LSPCLK_HZ;                        // set baud rate
178     clk /= baud*8;
179     clk--;
180     sci->SCILBAUD.all = clk & 0xFF;
181     sci->SCIHBAUD.all = (clk >> 8) & 0xFF;
182
183     sci->SCICCR.bit.SCICHAR = 0x7;          // (8) 8 bits per character
184     sci->SCICCR.bit.PARITYENA = 0;         // (N) disable party calculation
185     sci->SCICCR.bit.STOPBITS = 0;         // (1) transmit 1 stop bit
186     sci->SCICCR.bit.LOOPBKENA = 0;         // disable loopback test
187     sci->SCICCR.bit.ADDRIDLE_MODE = 0;     // idle-line mode (non-multiprocessor SCI comm)
188
189     sci->SCIFFCT.bit.FFTXDLY = 0;          // TX: zero-delay
190
191     sci->SCIFFTX.bit.SCIFFENA = 1;          // enable SCI fifo enhancements
192     sci->SCIFFTX.bit.TXFIFORESET = 0;
193     sci->SCIFFTX.bit.TXFFIL = 0x0; // TX: fifo interrupt at all levels      ??? is this correct
194     sci->SCIFFTX.bit.TXFFINTCLR = 1;       // TX: clear interrupt flag
195     sci->SCIFFTX.bit.TXFFIENA = 0;         // TX: disable fifo interrupt
196     sci->SCIFFTX.bit.TXFIFORESET = 1;
197
198     sci->SCIFFRX.bit.RXFIFORESET = 0;       // RX: fifo reset
199     sci->SCIFFRX.bit.RXFFINTCLR = 1;       // RX: clear interrupt flag
200     sci->SCIFFRX.bit.RXFFIENA = 1;         // RX: enable fifo interrupt
201     sci->SCIFFRX.bit.RXFFIL = 0x1;        // RX: fifo interrupt
202     sci->SCIFFRX.bit.RXFIFORESET = 1;      // RX: re-enable fifo
203
204     sci->SCICTL2.bit.RXBKINTENA = 0;        // disable receiver/error interrupt
205     sci->SCICTL2.bit.TXINTENA = 0;         // disable transmitter interrupt
206
207     sci->SCICTL1.bit.TXWAKE = 0;
208     sci->SCICTL1.bit.SLEEP = 0;            // disable sleep mode
209     sci->SCICTL1.bit.RXENA = 1;            // enable SCI receiver
210     sci->SCICTL1.bit.RXERRINTENA = 0;     // disable receive error interrupt
211     sci->SCICTL1.bit.TXENA = 1;            // enable SCI transmitter
212     sci->SCICTL1.bit.SWRESET = 1;         // re-enable SCI
213
214     /* enable PIE interrupts */
215     if (s == &SerialB) {
216         PieCtrlRegs.PIEIER9.bit.INTx3 = 1;
217         PieCtrlRegs.PIEIER9.bit.INTx4 = 1;
218         IER |= (M_INT9);
219         PieCtrlRegs.PIEACK.all = (PIEACK_GROUP9);
220     }
221
222     return 0;
223 }
224
225 void uninit_serialSCIB(serialSCIB_t *s)
226 {
227     volatile struct SCI_REGS *sci = s->sci;
228

```

```

229  /* disable PIE interrupts */
230  if (s == &SerialB) {
231      PieCtrlRegs.PIEIER9.bit.INTx3 = 0;
232      PieCtrlRegs.PIEIER9.bit.INTx4 = 0;
233      IER &= ~M_INT9;
234  }
235
236  sci->SCICTL1.bit.RXERRINTENA = 0;    // disable receive error interrupt
237  sci->SCICTL2.bit.RXBKINTENA = 0;    // disable receiver/error interrupt
238  sci->SCICTL2.bit.TXINTENA = 0;      // disable transmitter interrupt
239
240  sci->SCICTL1.bit.RXENA = 0;          // disable SCI receiver
241  sci->SCICTL1.bit.TXENA = 0;          // disable SCI transmitter
242 }
243
244
245
246 /*****
247  * SERIAL_SEND()
248  *
249  * "User level" function to send data via serial. Return value is the
250  * length of data successfully copied to the TX buffer.
251  *****/
252 uint16_t serial_sendSCIB(serialSCIB_t *s, char *data, uint16_t len)
253 {
254     uint16_t i = 0;
255     if (len && s->TX.size < BUF_SIZE_SCIB) {
256         for (i = 0; i < len; i++) {
257             if (buf_writeSCIB_1(&s->TX, data[i] & 0x00FF) != 0) break;
258         }
259         s->sci->SCIFFTX.bit.TXFFINTCLR = 1; // TX: clear interrupt flag
260         s->sci->SCIFFTX.bit.TXFFIENA = 1;   // TX: enable fifo interrupt
261     }
262     return i;
263 }
264
265 // for SerialC
266 uint16_t init_serialSCIC(serialSCIC_t *s, uint32_t baud)
267 {
268     volatile struct SCI_REGS *sci;
269     uint32_t clk;
270
271     if (s == &SerialC) {
272         sci = &ScicRegs;
273         s->sci = sci;
274         init_bufferSCIC(&s->TX);
275         GPIO_SetupPinMux(139, GPIO_MUX_CPU1, 6);
276         GPIO_SetupPinOptions(139, GPIO_INPUT, GPIO_PULLUP);
277         GPIO_SetupPinMux(56, GPIO_MUX_CPU1, 6);
278         GPIO_SetupPinOptions(56, GPIO_OUTPUT, GPIO_PUSHPULL);
279
280     } else {
281         return 1;
282     }
283
284     /* init for standard baud, 8N1 comm */
285     sci->SCICTL1.bit.SWRESET = 0;      // init SCI state machines and opt flags

```

```

286     sci->SCICCR.all = 0x0;
287     sci->SCICTL1.all = 0x0;
288     sci->SCICTL2.all = 0x0;
289     sci->SCIPRI.all = 0x0;
290     clk = LSPCLK_HZ;           // set baud rate
291     clk /= baud*8;
292     clk--;
293     sci->SCILBAUD.all = clk & 0xFF;
294     sci->SCIHBAUD.all = (clk >> 8) & 0xFF;
295
296     sci->SCICCR.bit.SCICHAR = 0x7;    // (8) 8 bits per character
297     sci->SCICCR.bit.PARITYENA = 0;    // (N) disable party calculation
298     sci->SCICCR.bit.STOPBITS = 0;    // (1) transmit 1 stop bit
299     sci->SCICCR.bit.LOOPBKENA = 0;    // disable loopback test
300     sci->SCICCR.bit.ADDRIDLE_MODE = 0; // idle-line mode (non-multiprocessor SCI comm)
301
302     sci->SCIFFCT.bit.FFTXDLY = 0;    // TX: zero-delay
303
304     sci->SCIFFTX.bit.SCIFFENA = 1;    // enable SCI fifo enhancements
305     sci->SCIFFTX.bit.TXFIFORESET = 0;
306     sci->SCIFFTX.bit.TXFFIL = 0x0; // TX: fifo interrupt at all levels      ??? is this correct
307     sci->SCIFFTX.bit.TXFFINTCLR = 1; // TX: clear interrupt flag
308     sci->SCIFFTX.bit.TXFFIENA = 0;    // TX: disable fifo interrupt
309     sci->SCIFFTX.bit.TXFIFORESET = 1;
310
311     sci->SCIFFRX.bit.RXFIFORESET = 0; // RX: fifo reset
312     sci->SCIFFRX.bit.RXFFINTCLR = 1;  // RX: clear interrupt flag
313     sci->SCIFFRX.bit.RXFFIENA = 1;    // RX: enable fifo interrupt
314     sci->SCIFFRX.bit.RXFFIL = 0x1;    // RX: fifo interrupt
315     sci->SCIFFRX.bit.RXFIFORESET = 1; // RX: re-enable fifo
316
317     sci->SCICTL2.bit.RXBKINTENA = 0;   // disable receiver/error interrupt
318     sci->SCICTL2.bit.TXINTENA = 0;    // disable transmitter interrupt
319
320     sci->SCICTL1.bit.TXWAKE = 0;
321     sci->SCICTL1.bit.SLEEP = 0;        // disable sleep mode
322     sci->SCICTL1.bit.RXENA = 1;        // enable SCI receiver
323     sci->SCICTL1.bit.RXERRINTENA = 0; // disable receive error interrupt
324     sci->SCICTL1.bit.TXENA = 1;        // enable SCI transmitter
325     sci->SCICTL1.bit.SWRESET = 1;     // re-enable SCI
326
327     /* enable PIE interrupts */
328     if (s == &SerialC) {
329         PieCtrlRegs.PIEIER8.bit.INTx5 = 1;
330         PieCtrlRegs.PIEIER8.bit.INTx6 = 1;
331         PieCtrlRegs.PIEACK.all = (PIEACK_GROUP8);
332         IER |= (M_INT8);
333     }
334
335     return 0;
336 }
337
338 void uninit_serialSCIC(serialSCIC_t *s)
339 {
340     volatile struct SCI_REGS *sci = s->sci;
341
342

```

```

343  /* disable PIE interrupts */
344  if (s == &SerialC) {
345      PieCtrlRegs.PIEIER8.bit.INTx5 = 0;
346      PieCtrlRegs.PIEIER8.bit.INTx6 = 0;
347      IER &= ~M_INT8;
348  }
349
350  sci->SCICTL1.bit.RXERRINTENA = 0;    // disable receive error interrupt
351  sci->SCICTL2.bit.RXBKINTENA = 0;    // disable receiver/error interrupt
352  sci->SCICTL2.bit.TXINTENA = 0;      // disable transmitter interrupt
353
354  sci->SCICTL1.bit.RXENA = 0;          // disable SCI receiver
355  sci->SCICTL1.bit.TXENA = 0;          // disable SCI transmitter
356 }
357
358
359
360 /*****
361  * SERIAL_SEND()
362  *
363  * "User level" function to send data via serial. Return value is the
364  * length of data successfully copied to the TX buffer.
365  *****/
366 uint16_t serial_sendSCIC(serialSCIC_t *s, char *data, uint16_t len)
367 {
368     uint16_t i = 0;
369     if (len && s->TX.size < BUF_SIZE_SCIC) {
370         for (i = 0; i < len; i++) {
371             if (buf_writeSCIC_1(&s->TX, data[i] & 0x00FF) != 0) break;
372         }
373         s->sci->SCIFFTX.bit.TXFFINTCLR = 1; // TX: clear interrupt flag
374         s->sci->SCIFFTX.bit.TXFFIENA = 1;   // TX: enable fifo interrupt
375     }
376     return i;
377 }
378
379 // For Serial D
380 uint16_t init_serialSCID(serialSCID_t *s, uint32_t baud)
381 {
382     volatile struct SCI_REGS *sci;
383     uint32_t clk;
384
385     if (s == &SerialD) {
386         sci = &ScidRegs;
387         s->sci = sci;
388         init_bufferSCID(&s->TX);
389         GPIO_SetupPinMux(105, GPIO_MUX_CPU1, 6);
390         GPIO_SetupPinOptions(105, GPIO_INPUT, GPIO_PULLUP);
391         GPIO_SetupPinMux(104, GPIO_MUX_CPU1, 6);
392         GPIO_SetupPinOptions(104, GPIO_OUTPUT, GPIO_PUSHPULL);
393     }
394     else {
395         return 1;
396     }
397
398     /* init for standard baud, 8N1 comm */
399     sci->SCICTL1.bit.SWRESET = 0;    // init SCI state machines and opt flags

```

```

400     sci->SCICCR.all = 0x0;
401     sci->SCICTL1.all = 0x0;
402     sci->SCICTL2.all = 0x0;
403     sci->SCIPRI.all = 0x0;
404     clk = LSPCLK_HZ;                // set baud rate
405     clk /= baud*8;
406     clk--;
407     sci->SCILBAUD.all = clk & 0xFF;
408     sci->SCIHBAUD.all = (clk >> 8) & 0xFF;
409
410     sci->SCICCR.bit.SCICHAR = 0x7;    // (8) 8 bits per character
411     sci->SCICCR.bit.PARITYENA = 0;    // (N) disable party calculation
412     sci->SCICCR.bit.STOPBITS = 0;    // (1) transmit 1 stop bit
413     sci->SCICCR.bit.LOOPBKENA = 0;    // disable loopback test
414     sci->SCICCR.bit.ADDRIDLE_MODE = 0; // idle-line mode (non-multiprocessor SCI comm)
415
416     sci->SCIFFCT.bit.FFTXDLY = 0;    // TX: zero-delay
417
418     sci->SCIFFTX.bit.SCIFFENA = 1;    // enable SCI fifo enhancements
419     sci->SCIFFTX.bit.TXFIFORESET = 0;
420     sci->SCIFFTX.bit.TXFFIL = 0x0; // TX: fifo interrupt at all levels      ??? is this correct
421     sci->SCIFFTX.bit.TXFFINTCLR = 1; // TX: clear interrupt flag
422     sci->SCIFFTX.bit.TXFFIENA = 0;    // TX: disable fifo interrupt
423     sci->SCIFFTX.bit.TXFIFORESET = 1;
424
425     sci->SCIFFRX.bit.RXFIFORESET = 0; // RX: fifo reset
426     sci->SCIFFRX.bit.RXFFINTCLR = 1;  // RX: clear interrupt flag
427     sci->SCIFFRX.bit.RXFFIENA = 1;    // RX: enable fifo interrupt
428     sci->SCIFFRX.bit.RXFFIL = 0x1;    // RX: fifo interrupt
429     sci->SCIFFRX.bit.RXFIFORESET = 1; // RX: re-enable fifo
430
431     sci->SCICTL2.bit.RXBKINTENA = 0;   // disable receiver/error interrupt
432     sci->SCICTL2.bit.TXINTENA = 0;    // disable transmitter interrupt
433
434     sci->SCICTL1.bit.TXWAKE = 0;
435     sci->SCICTL1.bit.SLEEP = 0;        // disable sleep mode
436     sci->SCICTL1.bit.RXENA = 1;        // enable SCI receiver
437     sci->SCICTL1.bit.RXERRINTENA = 0;  // disable receive error interrupt
438     sci->SCICTL1.bit.TXENA = 1;        // enable SCI transmitter
439     sci->SCICTL1.bit.SWRESET = 1;      // re-enable SCI
440
441     /* enable PIE interrupts */
442     if (s == &SerialD) {
443         PieCtrlRegs.PIEIER8.bit.INTx7 = 1;
444         PieCtrlRegs.PIEIER8.bit.INTx8 = 1;
445         PieCtrlRegs.PIEACK.all = (PIEACK_GROUP8);
446         IER |= (M_INT8);
447     }
448
449     return 0;
450 }
451
452 void uninit_serialSCID(serialSCID_t *s)
453 {
454     volatile struct SCI_REGS *sci = s->sci;
455
456     /* disable PIE interrupts */

```



```

457     if (s == &SerialD) {
458         PieCtrlRegs.PIEIER8.bit.INTx7 = 0;
459         PieCtrlRegs.PIEIER8.bit.INTx8 = 0;
460         IER &= ~M_INT8;
461     }
462
463     sci->SCICTL1.bit.RXERRINTENA = 0;    // disable receive error interrupt
464     sci->SCICTL2.bit.RXBKINTENA = 0;    // disable receiver/error interrupt
465     sci->SCICTL2.bit.TXINTENA = 0;    // disable transmitter interrupt
466
467     sci->SCICTL1.bit.RXENA = 0;    // disable SCI receiver
468     sci->SCICTL1.bit.TXENA = 0;    // disable SCI transmitter
469 }
470
471
472
473 /*****
474  * SERIAL_SEND()
475  *
476  * "User level" function to send data via serial.  Return value is the
477  * length of data successfully copied to the TX buffer.
478  *****/
479 uint16_t serial_sendSCID(serialSCID_t *s, char *data, uint16_t len)
480 {
481     uint16_t i = 0;
482     if (len && s->TX.size < BUF_SIZE_SCID) {
483         for (i = 0; i < len; i++) {
484             if (buf_writeSCID_1(&s->TX, data[i] & 0x00FF) != 0) break;
485         }
486         s->sci->SCIFFTX.bit.TXFFINTCLR = 1;    // TX: clear interrupt flag
487         s->sci->SCIFFTX.bit.TXFFIENA = 1;    // TX: enable fifo interrupt
488     }
489     return i;
490 }
491
492
493
494 /*****
495  * TXxINT_DATA_SENT()
496  *
497  * Executed when transmission is ready for additional data.  These functions
498  * read the next char of data and put it in the TXBUF register for transfer.
499  *****/
500 #ifdef _FLASH
501 #pragma CODE_SECTION(TXAINT_data_sent, ".TI.ramfunc");
502 #endif
503 __interrupt void TXAINT_data_sent(void)
504 {
505     char data;
506     if (buf_readSCIA_1(&SerialA.TX, 0, &data) == 0) {
507         while ( (buf_readSCIA_1(&SerialA.TX, 0, &data) == 0)
508             && (SerialA.sci->SCIFFTX.bit.TXFFST != 0x10) ) {
509             buf_removeSCIA(&SerialA.TX, 1);
510             SerialA.sci->SCITXBUF.all = data;
511         }
512     } else {
513         SerialA.sci->SCIFFTX.bit.TXFFIENA = 0;    // TX: disable fifo interrupt

```

```

514     }
515     SerialA.sci->SCIFFTX.bit.TXFFINTCLR = 1; // TX: clear interrupt flag
516     PieCtrlRegs.PIEACK.all = PIEACK_GROUP9;
517 }
518
519
520 //for serialB
521 #ifdef _FLASH
522 #pragma CODE_SECTION(TXBINT_data_sent, ".TI.ramfunc");
523 #endif
524 __interrupt void TXBINT_data_sent(void)
525 {
526     char data;
527     if (buf_readSCIB_1(&SerialB.TX,0,&data) == 0) {
528         while ( (buf_readSCIB_1(&SerialB.TX,0,&data) == 0)
529             && (SerialB.sci->SCIFFTX.bit.TXFFST != 0x10) ) {
530             buf_removeSCIB(&SerialB.TX, 1);
531             SerialB.sci->SCITXBUF.all = data;
532         }
533     } else {
534         SerialB.sci->SCIFFTX.bit.TXFFIENA = 0; // TX: disable fifo interrupt
535     }
536     SerialB.sci->SCIFFTX.bit.TXFFINTCLR = 1; // TX: clear interrupt flag
537     PieCtrlRegs.PIEACK.all = PIEACK_GROUP9;
538 }
539
540
541 //for serialC
542 #ifdef _FLASH
543 #pragma CODE_SECTION(TXCINT_data_sent, ".TI.ramfunc");
544 #endif
545 __interrupt void TXCINT_data_sent(void)
546 {
547     char data;
548     if (buf_readSCIC_1(&SerialC.TX,0,&data) == 0) {
549         while ( (buf_readSCIC_1(&SerialC.TX,0,&data) == 0)
550             && (SerialC.sci->SCIFFTX.bit.TXFFST != 0x10) ) {
551             buf_removeSCIC(&SerialC.TX, 1);
552             SerialC.sci->SCITXBUF.all = data;
553         }
554     } else {
555         SerialC.sci->SCIFFTX.bit.TXFFIENA = 0; // TX: disable fifo interrupt
556     }
557     SerialC.sci->SCIFFTX.bit.TXFFINTCLR = 1; // TX: clear interrupt flag
558     PieCtrlRegs.PIEACK.all = PIEACK_GROUP8;
559 }
560
561 //for serialD
562 #ifdef _FLASH
563 #pragma CODE_SECTION(TXDINT_data_sent, ".TI.ramfunc");
564 #endif
565 __interrupt void TXDINT_data_sent(void)
566 {
567     char data;
568     if (buf_readSCID_1(&SerialD.TX,0,&data) == 0) {
569         while ( (buf_readSCID_1(&SerialD.TX,0,&data) == 0)
570             && (SerialD.sci->SCIFFTX.bit.TXFFST != 0x10) ) {

```

```

571         buf_removeSCID(&SerialD.TX, 1);
572         SerialD.sci->SCITXBUF.all = data;
573     }
574 } else {
575     SerialD.sci->SCIFFTX.bit.TXFFIENA = 0;    // TX: disable fifo interrupt
576 }
577 SerialD.sci->SCIFFTX.bit.TXFFINTCLR = 1;    // TX: clear interrupt flag
578 PieCtrlRegs.PIEACK.all = PIEACK_GROUP8;
579 }
580
581 //for SerialA
582 #ifdef _FLASH
583 #pragma CODE_SECTION(RXAINT_recv_ready, ".TI.ramfunc");
584 #endif
585 __interrupt void RXAINT_recv_ready(void)
586 {
587     RXAdata = SciaRegs.SCIRXBUF.all;
588
589     /* SCI PE or FE error */
590     if (RXAdata & 0xC000) {
591         SciaRegs.SCICTL1.bit.SWRESET = 0;
592         SciaRegs.SCICTL1.bit.SWRESET = 1;
593         SciaRegs.SCIFFRX.bit.RXFIFORESET = 0;
594         SciaRegs.SCIFFRX.bit.RXFIFORESET = 1;
595     } else {
596         RXAdata = RXAdata & 0x00FF;
597         if(RXAdata == 'a') {
598             GpioDataRegs.GPBCLEAR.bit.GPIO34 = 1;
599         } else if(RXAdata == 's') {
600             GpioDataRegs.GPBSET.bit.GPIO34 = 1;
601         }
602         if(RXAdata == 'z') {
603             GpioDataRegs.GPASET.bit.GPIO22 = 1;
604         } else if(RXAdata == 'x') {
605             GpioDataRegs.GPACLEAR.bit.GPIO22 = 1;
606         }
607         if(RXAdata == 'c') {
608             GpioDataRegs.GPCSET.bit.GPIO94 = 1;
609         } else if(RXAdata == 'v') {
610             GpioDataRegs.GPCCLEAR.bit.GPIO94 = 1;
611         }
612         if(RXAdata == 'b') {
613             GpioDataRegs.GPCSET.bit.GPIO95 = 1;
614         } else if(RXAdata == 'n') {
615             GpioDataRegs.GPCCLEAR.bit.GPIO95 = 1;
616         }
617         if(RXAdata == 'm') {
618             GpioDataRegs.GPDSET.bit.GPIO97 = 1;
619         } else if(RXAdata == ',') {
620             GpioDataRegs.GPDCLEAR.bit.GPIO97 = 1;
621         }
622
623         numRXA ++;
624     }
625
626     SciaRegs.SCIFFRX.bit.RXFFINTCLR = 1;
627     PieCtrlRegs.PIEACK.all = PIEACK_GROUP9;

```

```
628
629 }
630
631 //for SerialB
632 #ifdef _FLASH
633 #pragma CODE_SECTION(RXBINT_recv_ready, ".TI.ramfunc");
634 #endif
635 __interrupt void RXBINT_recv_ready(void)
636 {
637     RXBdata = ScibRegs.SCIRXBUF.all;
638
639     /* SCI PE or FE error */
640     if (RXBdata & 0xC000) {
641         ScibRegs.SCICTL1.bit.SWRESET = 0;
642         ScibRegs.SCICTL1.bit.SWRESET = 1;
643         ScibRegs.SCIFFRX.bit.RXFIFORESET = 0;
644         ScibRegs.SCIFFRX.bit.RXFIFORESET = 1;
645     } else {
646         RXBdata = RXBdata & 0x00FF;
647         // Do something with recieved character
648         numRXB ++;
649     }
650
651     ScibRegs.SCIFFRX.bit.RXFFINTCLR = 1;
652     PieCtrlRegs.PIEACK.all = PIEACK_GROUP9;
653 }
654
655
656 // for SerialC
657 #ifdef _FLASH
658 #pragma CODE_SECTION(RXCINT_recv_ready, ".TI.ramfunc");
659 #endif
660 __interrupt void RXCINT_recv_ready(void)
661 {
662     RXCdata = ScicRegs.SCIRXBUF.all;
663
664     /* SCI PE or FE error */
665     if (RXCdata & 0xC000) {
666         ScicRegs.SCICTL1.bit.SWRESET = 0;
667         ScicRegs.SCICTL1.bit.SWRESET = 1;
668         ScicRegs.SCIFFRX.bit.RXFIFORESET = 0;
669         ScicRegs.SCIFFRX.bit.RXFIFORESET = 1;
670     } else {
671         RXCdata = RXCdata & 0x00FF;
672         numRXC ++;
673     }
674 }
675
676     ScicRegs.SCIFFRX.bit.RXFFINTCLR = 1;
677     PieCtrlRegs.PIEACK.all = PIEACK_GROUP8;
678 }
679
680
681 //for SerialD
682 #ifdef _FLASH
683 #pragma CODE_SECTION(RXDINT_recv_ready, ".TI.ramfunc");
684 #endif
```

```

685 __interrupt void RXDINT_recv_ready(void)
686 {
687     RXDdata = ScidRegs.SCIRXBUF.all;
688
689     /* SCI PE or FE error */
690     if (RXDdata & 0xC000) {
691         ScidRegs.SCICTL1.bit.SWRESET = 0;
692         ScidRegs.SCICTL1.bit.SWRESET = 1;
693         ScidRegs.SCIFFRX.bit.RXFIFORESET = 0;
694         ScidRegs.SCIFFRX.bit.RXFIFORESET = 1;
695     } else {
696         RXDdata = RXDdata & 0x00FF;
697         numRXD ++;
698     }
699 }
700
701 ScidRegs.SCIFFRX.bit.RXFFINTCLR = 1;
702 PieCtrlRegs.PIEACK.all = PIEACK_GROUP8;
703 }
704
705
706 // SerialA only setup for Tera Term connection
707 char serial_printf_bufSCIA[BUF_SIZE_SCIA];
708
709 uint16_t serial_printf(serialSCIA_t *s, char *fmt, ...)
710 {
711     va_list ap;
712
713     va_start(ap, fmt);
714     vsprintf(serial_printf_bufSCIA, fmt, ap);
715     va_end(ap);
716
717     return serial_sendSCIA(s, serial_printf_bufSCIA, strlen(serial_printf_bufSCIA));
718 }
719
720 //For Text LCD
721 char UART_printf_buffer[BUF_SIZE_SCIB];
722 void UART_vprintfLine(unsigned char line, char *format, va_list ap)
723 {
724     char sendmsg[24];
725
726     int i;
727
728     vsprintf(UART_printf_buffer, format, ap);
729
730     // Add header information and pad end of transfer with spaces to clear display
731     sendmsg[0] = 0xFE;
732     sendmsg[1] = 'G';
733     sendmsg[2] = 1;
734     sendmsg[3] = line;
735     for (i=4; i<24; i++) {
736         if (i >= strlen(UART_printf_buffer)+4) {
737             sendmsg[i] = ' ';
738         } else {
739             sendmsg[i] = UART_printf_buffer[i-4];
740         }
741     }

```

```
742    serial_sendSCIB(&SerialB,sendmsg,24);
743 }
744
745 void UART_printfLine(unsigned char line, char *format, ...)
746 {
747     va_list ap;
748     va_start(ap, format);
749     UART_vprintfLine(line,format,ap);
750 }
751
752
```