

DSP/BIOS Example

TSK Level

Arrows in the TSK and between the TSK and HWI Levels indicate direction of multi-thread communication.

User Created PRDs, SWIs or TSKs call for example SendWireless(..) to send a message to the PC. SendWireless, places the message in the Queue, SendStrmsgQueue, and the Activates the Semaphore, SEM_SendStrmsg_rdy.

Semaphore: SEM_SendStrmsg_rdy
Queue: SendStrmsgQueue

User Defined Receive TSK waits for a new character by suspending (or blocking) itself until the semaphore, SEM_UART1RecChar_rdy is set active by HWI 4's function. The new character is then read from the Queue, UART1RecCharQueue. After receiving this character, the task loops back to the beginning of its code and again blocks itself to wait for the next character to be sent.

Semaphore: SEM_UART1RecChar_rdy
Queue: UART1RecCharQueue

TSK_UART
Function: uarttsk

- Blocks itself from running until the semaphore, SEM_SendStrmsg_rdy, is set active.
- Fills global array "txbuffer" with the characters given in the Queue, SendStrmsgQueue.
- Enables Timer1 to Send the Characters out at the specified baud rate.
- Blocks itself from running until the Semaphore, SEM_SendStrmsg_done, is set active.
- Loops back to the top to wait for next message to send.

HWI Level

Global char Array: txbuffer

Semaphore: SEM_SendStrmsg_done

HWI_INT4
Function: extint4_cisr
Receives an interrupt from the MAX3100 chip when there is a new character received on the RS-232 side of the serial port.

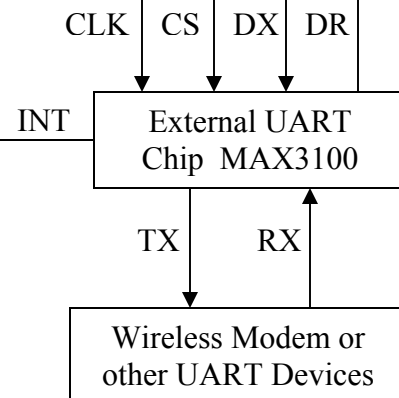
HWI_INT15
Function: timer1_cisr
Timer 1 is setup to send out a new 8 bit character on the SPI serial port at the specified baud rate.

Read SPI registers Write to SPI registers

DSP's SPI (McBSP)
Serial Port

Hardware Level

Arrows in the Hardware Level indicate actual signals and their Direction.



RS-232 standard serial port. (The standard serial port on the back of you PC.)

```

1 CHARACTER RECEIVE CODE PAGE 1 of 4
2
3 // Receive Character QUEUE structure, passing in the QUEUE a character sent on the serial port.
4 typedef struct RecCharQueObj {
5     QUE_Element_t elem;
6     char recchar;
7 } RecCharQueObj;
8
9 // Function: void Init_UART1and2(int BaudRate)
10 // Parameters: BaudRate choice. Only three options currently supported 0 = 19200 Baud both UARTS
11 // 1 = 57600 Baud both UARTS
12 // 2 = 115200 Baud both UARTS
13 // Note that UART1 is always configured as 57600 baud since it is wired to the wireless modem
14 // Return value: None
15 // Description: Initializes the serial port to send data to both UART1 and UART2.
16 // Must be called before any UART command
17 void Init_UART1and2(int BaudRate) {
18
19     RecCharQueObj *RecChar_que_setup;
20     ***** Other Code *****
21     .....
22     *****
23     // setup Queues and initial UART1 messages
24     RecChar_que_setup = (RecCharQueObj *)MEM_malloc(SDRAM, NUM_RECEIVE_QUEUES * sizeof(RecCharQueObj), 0);
25
26     if (RecChar_que_setup == MEM_ILLEGAL) {
27         SYS_abort("Memory allocation failed!\n");
28     }
29
30     // put all allocated memory into the "free" Queue
31     for (i=0; i < NUM_RECEIVE_QUEUES; i++, RecChar_que_setup++) {
32         QUE_put(&UART1RecCharfreeQueue, RecChar_que_setup);
33     }
34
35     ***** Other Code *****
36     .....
37     *****
38
39 }
40
41
42
43
44
45
46

```

```
47 CHARACTER RECEIVE CODE PAGE 2 of 4
48
49 // UART 1 GLOBAL VARIABLES
50 extern far SEM_Obj SEM_UART1MessageReady;
51 char UART1receivechar;
52 int UART1beginnewdata = 0;
53 int UART1datacollected = 0;
54 char UART1MessageArray[400];
55
56 void UART1ReceiveCharTask(void) {
57
58     while(1) { // Loop forever
59         // Wait forever for new character
60         if (checkfornewcharUART1(&UART1receivechar, SYS_FOREVER) ) {
61             if (!UART1beginnewdata) { // Only true if have not yet begun a message
62                 if ((unsigned char)UART1receivechar == 253) { // Check for start char
63                     if (SEM_count(&SEM_UART1MessageReady) == 0) { // Last data sent was used if = 0
64                         UART1datacollected = 0; // amount of data collected in message set to 0
65                         UART1beginnewdata = 1; // flag to indicate we are collecting a message
66                     }
67                 }
68             } else { // Filling data
69                 // Dont go too large... limit message to 30 chars
70                 if ((UART1datacollected < 400) && ((unsigned char)UART1receivechar != 255)) {
71                     UART1MessageArray[UART1datacollected] = (char) UART1receivechar;
72                     UART1datacollected++;
73                 } else { // too much data or 255 char received
74                     if ((unsigned char)UART1receivechar != 255) {
75                         UART1datacollected = 0;
76                         UART1beginnewdata = 0;
77                     } else {
78                         UART1MessageArray[UART1datacollected] = '\0'; // Null terminate the string
79                         SEM_post(&SEM_UART1MessageReady);
80                         UART1beginnewdata = 0;
81                         UART1datacollected = 0;
82                     }
83                 }
84             }
85         }
86     }
87 }
88
89
90
91
92
```

```

93 CHARACTER RECEIVE CODE PAGE 3 of 4
94
95 // Function: Bool checkfornewcharUART1(char* newchar, Uns timeout)
96 // Parameters:
97 // ->newchar -> A pointer to where the new character should be stored.
98 // ->timeout -> Number of clock cycles to wait for the SEM_UART1RecChar_rdy semaphore.
99 // Return value: -> Returns TRUE if successfully received a character, FALSE otherwise.
100 Bool checkfornewcharUART1(char* newchar, Uns timeout) {
101     ->
102     ->RecCharQueObj *charmsg;
103     ->
104     ->if (SEM_pend(&SEM_UART1RecChar_rdy, timeout)) {
105         ->charmsg = QUE_get(&UART1RecCharQueue);
106         ->*newchar = charmsg->recchar;
107         ->QUE_put(&UART1RecCharFreeQueue, charmsg);
108         ->return(TRUE);
109     } else {
110         ->*newchar = '\0';
111         ->return(FALSE);
112     }
113 }
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138

```

```
139 CHARACTER RECEIVE CODE PAGE 4 of 4
140
141 void exti nt4_cisr(void) {
142     →
143     →char recchar;
144     →int rawrecdata;
145
146     →SelectCOM1();
147     →*(unsigned volatile int *)McBSP1_DXR = 0x0000; // clear int
148     →while ((*unsigned volatile int *)McBSP1_SPCR & 0x20002) != 0x20002) {} // wait for transmit complete
149     →rawrecdata = *(unsigned volatile int *)McBSP1_DRR;
150     →
151     →if ((rawrecdata & 0x8000) == 0x8000) { // then there is a new char in rawrecdata
152     →→recchar = (char) (rawrecdata & 0xFF);
153     →→sendnewcharUART1(recchar);
154     →} else {
155     →→UART1fantomint++;
156     →}
157     →
158 }
159
160 void sendnewcharUART1(char newchar) {
161     →
162     →RecCharQueObj *charmsg;
163
164     →// if the Que is full (i.e. freeQue empty) don't send the message... in the future should return an error
165     →if (!QUE_empty(&UART1RecCharFreeQueue)) {
166     →→charmsg = QUE_get(&UART1RecCharFreeQueue);
167
168     →→charmsg->recchar = newchar;
169
170     →→QUE_put(&UART1RecCharQueue, charmsg);
171
172     →→SEM_post(&SEM_UART1RecChar_rdy);
173     →}
174
175 }
176
177
178
179
180
181
182
183
184
```

```
185 CHARACTER ARRAY SEND CODE PAGE 1 of 4
186
187 // Function: void WirelessSend(char *buffer, int size)
188 // Parameters:
189 //   buffer -> A pointer to the character array to be sent to UART 1, Wireless modem.
190 //   size -> The number of characters to send from the array buffer. Size limited 60 Characters
191 // Return value: None
192 // Description: Sends string to UART1
193 void WirelessSend(char *buffer, int size) {
194
195     int i;
196     unsigned char sendmsg[142];
197
198     sendmsg[0] = 253;
199     if (size < 140) {
200         for (i=0; i<size; i++) {
201             sendmsg[i+1] = (unsigned char) buffer[i];
202         }
203         sendmsg[i+1] = 255;
204         SendStr2_UART1((char*)sendmsg, size+2);
205     } else {
206         for (i=0; i<140; i++) {
207             sendmsg[i+1] = (unsigned char) buffer[i];
208         }
209         sendmsg[i+1] = 255;
210         SendStr2_UART1((char*)sendmsg, 142);
211     }
212
213 }
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
```

```
231
232 CHARACTER ARRAY SEND CODE PAGE 2 of 4
233
234 // Function: void SendStr2UART1(char *buffer, int size)
235 // Parameters:
236 // → buffer → A pointer to the character array to be sent to UART 1.
237 // → size → The number of characters to send from the array buffer.
238 // Return value: None
239 // Description: Starts a task to send a string of characters to UART 1.
240 void SendStr2_UART1(char *buffer, int size) {
241     →
242     → int putserr = 0;
243     → SendStrQueueObj *UARTmsg;
244     → int i;
245     →
246     → // Limit size of character transmission to MAX_SEND_LENGTH characters.
247     → if (size > MAX_SEND_LENGTH) size = MAX_SEND_LENGTH;
248     → if (size < 1) putserr = 1;
249
250     → if (putserr == 0) {
251     →     → // if the Que is full (i.e. freeQue empty) don't send the message... in the future should return an error
252     →     → if (!QUE_empty(&SendStrfreeQueue)) {
253     →     →     → UARTmsg = QUE_get(&SendStrfreeQueue);
254
255     →     →     → UARTmsg->msg_COMPOR = 1;
256     →     →     → UARTmsg->msg_len = size;
257     →     →     → for (i=0; i < size; i++) {
258     →     →     →     → UARTmsg->msg[i] = buffer[i];
259     →     →     → }
260     →     →     →
261     →     →     → QUE_put(&SendStrmsgQueue, UARTmsg);
262
263     →     →     → SEM_post(&SEM_SendStrmsg_rdy);
264     →     →     → }
265     →     → }
266
267 }
268
269
270
271
272
273
274
275
276
```

```

277 CHARACTER ARRAY SEND CODE PAGE 3 of 4
278
279 // Function uarttsk,
280 // This function is a task that is used to send character messages to the
281 // Matrix Orbital LCD panel. It is in a continuous while loop first waiting
282 // for a Semaphore post from the function lcdputs. When a semaphore arrives
283 // the LCDmsgQueue is read to get the message to send to the LCD. The EDMA is
284 // setup to send the message and then the task wait again for another semaphore
285 // post from the EDMA's interrupt service routine (edma_cisr) indicating that the transfer
286 // has been completed. Then lcdprinttsk goes back to waiting for the next lcdputs
287 // post.
288 void uarttsk(void) {
289     SendStrQueObj *sendmsg;
290     int i;
291     while(1) { // Loop forever
292         SEM_pend(&SEM_SendStrmsg_rdy, SYS_FOREVER); // wait until LCD msg sent
293         sendmsg = QUE_get(&SendStrmsgQueue);
294         if (txserialmsg == 0) { // this should always be zero but in case of error do not
295             txlength = sendmsg->msg_len;
296             txCOM = sendmsg->msg_COMPOR;
297             for(i=0; i<txlength; i++) {
298                 txbuffer[i] = sendmsg->msg[i];
299             }
300             *(unsigned volatile int *)TIMER1_CTRL = 0x201; // initialize the timer1
301             //
302             *(unsigned volatile int *)TIMER1_PRD = 9766; // set timer1's period register to .000174s which is the time to send one char
303             //
304             ICR = 0x8000; // clear pending ints
305             IER |= 0x8000; // enable timer1 int
306             //
307             *(unsigned volatile int *)TIMER1_CTRL = 0x2C1; // start timer1
308             //
309             txserialmsg = 1;
310             //
311             SEM_pend(&SEM_SendStrmsg_done, SYS_FOREVER);
312             //
313             }
314         QUE_put(&SendStrfreeQueue, sendmsg);
315     }
316 }
317
318
319
320
321

```

```

322 CHARACTER ARRAY SEND CODE PAGE 4 of 4
323
324 int need_to_check_CTS = 1;
325 int CTScount = 0;
326 void timer1_isr(void) {
327
328     char recchar;
329     int rawrecdata;
330
331     if (txserialmsg == 1) {
332         if (chars_xmitted == txlength) {
333             txserialmsg = 0;
334             chars_xmitted = 0;
335             IER &= 0x7FFF; // disable timer1 int
336             SEM_post(&SEM_SendStrmsg_done);
337         } else {
338             SelectCOM1();
339             if (need_to_check_CTS == 0) {
340                 *(unsigned volatile int *)McBSP1_DXR = 0x0000; // Read the status of CTS
341                 while ((*unsigned volatile int *)McBSP1_SPCR & 0x2) != 0x2) {} // wait for transmit complete
342                 rawrecdata = *(unsigned volatile int *)McBSP1_DRR; // read CTS status and possibly a received char
343                 if ((rawrecdata & 0x8000) == 0x8000) { // then there is a new char in rawrecdata
344                     recchar = (char) (rawrecdata & 0xFF);
345                     sendnewcharUART1(recchar);
346                 }
347             }
348             if ((need_to_check_CTS != 0) || ((rawrecdata & 0x0200) == 0x200)) { // check CTS low
349                 *(unsigned volatile int *)McBSP1_DXR = (unsigned short)((unsigned short)txbuffer[chars_xmitted] | 0x8000) & 0x80FF);
350                 chars_xmitted++;
351                 need_to_check_CTS = chars_xmitted%64;
352                 while ((*unsigned volatile int *)McBSP1_SPCR & 0x20002) != 0x20002) {} // wait for transmit complete
353                 rawrecdata = *(unsigned volatile int *)McBSP1_DRR;
354
355                 if ((rawrecdata & 0x8000) == 0x8000) { // then there is a new char in rawrecdata
356                     recchar = (char) (rawrecdata & 0xFF);
357                     sendnewcharUART1(recchar);
358                 }
359             } else {
360                 CTScount++;
361             }
362         }
363     }
364 }

```