

```

1  /*****
2  *
3  *   _____   _____   _____   _____   _____   _____
4  *   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
5  *   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
6  *   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
7  *   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
8  *   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
9  *****/
10 *
11 * File      : esp.c
12 * Version   : 1.0
13 *
14 *****/
15 *
16 * Description : Managing ESP32 state machine and commands
17 *
18 *****/
19 *
20 * Author      : Miguel Santos
21 * Date        : 25.09.2023
22 *
23 *****/
24 *
25 * MPLAB X      : 5.45
26 * XC32         : 2.50
27 * Harmony      : 2.06
28 *
29 *****/
30
31 #include "esp.h"
32
33 /*****/
34
35 /* Enable or disable debug of specific parts by (un)comment */
36 #ifndef DEBUG_LED
37     #define DEBUG_LED PORTGbits.RG9
38 #endif
39 // #define DEBUG_ESP_UART
40 // #define DEBUG_ESP_GET
41 // #define DEBUG_ESP_SEND
42
43 /*****/
44
45 /* Define UART and interrupts used by ESP32 */
46 #define ESP_USART_ID          USART_ID_1
47 #define ESP_INT_SOURCE_USART_ERROR    INT_SOURCE_USART_1_ERROR
48 #define ESP_INT_SOURCE_USART_RECEIVE  INT_SOURCE_USART_1_RECEIVE
49 #define ESP_INT_SOURCE_USART_TRANSMIT INT_SOURCE_USART_1_TRANSMIT
50
51 /*****/
52
53 /* Time to wait for fifo to be filled */
54 #define ESP_COUNT_RECEIVE_MS 20
55
56 /* Time waiting for a response of main app */
57 #define ESP_COUNT_WAIT_MS 2000
58
59 /*****/
60
61 /* Declaration of global application data */
62 ESP_DATA espData;
63
64 /*****/
65
66 /**
67  * @brief ESP_Initialize
68  *
69  * Initialize ESP32 state machine, counters and FIFOs
70  *
71  * @param void
72  * @return void
73  */

```

```

74 void ESP_Initialize ( void )
75 {
76     /* Place the App state machine in its default state. */
77     espData.state = ESP_STATE_IDLE;
78
79     /* Initial flags values */
80     espData.transmit = false;
81     espData.receive = false;
82
83     /* Initialize timing counters */
84     CNT_Initialize(&espData.cntReceive, ESP_COUNT_RECEIVE_MS);
85     CNT_Initialize(&espData.cntWait, ESP_COUNT_WAIT_MS);
86
87     /* Initialize FIFO descriptors */
88     FIFO_Initialize(&espData.fifoDesc_tx, ESP_FIFO_SIZE,
89                   espData.fifoBuff_tx, 0x00);
90     FIFO_Initialize(&espData.fifoDesc_rx, ESP_FIFO_SIZE,
91                   espData.fifoBuff_rx, 0x00);
92 }
93
94 /*****
95
96 /**
97  * @brief ESP_Tasks
98  *
99  * Execute ESP32 state machine, should be called cyclically
100  *
101  * @param void
102  * @return void
103  */
104 void ESP_Tasks ( void )
105 {
106     /* Check current state. */
107     switch ( espData.state )
108     {
109         /* Waiting for next state */
110         case ESP_STATE_IDLE:
111         {
112             /* Something to transmit in FIFO */
113             if(espData.transmit)
114             {
115                 espData.state = ESP_STATE_TRANSMIT;
116             }
117             /* Something to receive in FIFO */
118             else if(espData.receive)
119             {
120                 CNT_Reset(&espData.cntReceive);
121                 espData.state = ESP_STATE_RECEIVE;
122             }
123             break;
124         }
125
126         /* A new message has to be transmitted */
127         case ESP_STATE_TRANSMIT:
128         {
129             /* Check if data still available in FIFO */
130             if(FIFO_GetReadSpace(&espData.fifoDesc_tx))
131             {
132                 /* Enable the interrupt if needed */
133                 if(!SYS_INT_SourceIsEnabled(ESP_INT_SOURCE_USART_TRANSMIT))
134                 {
135                     SYS_INT_SourceEnable(ESP_INT_SOURCE_USART_TRANSMIT);
136                 }
137             }
138             else
139             {
140                 /* Leave state when fifo is empty */
141                 espData.transmit = false;
142                 espData.state = ESP_STATE_IDLE;
143             }
144             break;
145         }
146     }

```

```

147     /* Receiving a message by UART */
148     case ESP_STATE_RECEIVE:
149     {
150         /* Waiting for fifo to be filled */
151         if(CNT_Check(&espData.cntReceive))
152         {
153             if(FIFO_GetBuffer(&espData.fifoDesc_rx, (uint8_t*)&espData.resBuffer))
154             {
155                 /* Get first line of message */
156                 espData.p_resBuffer = strtok(espData.resBuffer, "\r\n");
157
158                 /* Flag state */
159                 espData.translate = true;
160
161                 /* Change directly to translate state */
162                 espData.state = ESP_STATE_TRANSLATE;
163             }
164             else
165             {
166                 /* An error occurred, going back to IDLE */
167                 espData.state = ESP_STATE_IDLE;
168             }
169
170             /* Flag state */
171             espData.receive = false;
172         }
173         break;
174     }
175
176     /* Translating different parts of the message received */
177     case ESP_STATE_TRANSLATE:
178     {
179         /* A command is detected */
180         if(espData.p_resBuffer[0] == 'A' && espData.p_resBuffer[1] == 'T')
181         {
182             strcpy(espData.atResponse.command, espData.p_resBuffer);
183         }
184         /* Acknowledge detected */
185         if(strcmp(espData.p_resBuffer, AT_ACK_OK) ||
186            strcmp(espData.p_resBuffer, AT_ACK_ERROR))
187         {
188             strcpy(espData.atResponse.ack, espData.p_resBuffer);
189         }
190         /* Data is detected */
191         else
192         {
193             strcpy(espData.atResponse.data, espData.p_resBuffer);
194         }
195
196         /* Get next line in string */
197         espData.p_resBuffer = strtok(NULL, "\r\n");
198
199         /* Leave state when no more lines */
200         if(espData.p_resBuffer == NULL)
201         {
202             /* Reset buffer */
203             memset(espData.resBuffer, 0x00, sizeof(espData.resBuffer));
204
205             /* Machine states and flags */
206             espData.newMessage = true;
207             espData.translate = false;
208             espData.wait = true;
209             espData.state = ESP_STATE_WAIT;
210
211             CNT_Reset(&espData.cntWait);
212         }
213         break;
214     }
215
216     /* Waiting for main application to answer */
217     case ESP_STATE_WAIT:
218     {
219         if(CNT_Check(&espData.cntWait))

```

```

220         {
221             espData.newMessage = false;
222             espData.state = ESP_STATE_IDLE;
223         }
224         break;
225     }
226
227     /* The default state should never be executed. */
228     default:
229     {
230         /* TODO: Handle error in application's state machine. */
231         break;
232     }
233 }
234 }
235
236 /*****
237
238 /**
239  * @brief ESP_SendCommand
240  *
241  * Send a command to the ESP32, managed by state machine
242  *
243  * @param char Command to send ; Use constant definitions
244  * @return bool True = command send ; False = Not allowed to send a command
245  */
246 bool ESP_SendCommand( char *p_command )
247 {
248     /* Local variables */
249     S_Fifo *p_fifoDesc;
250     uint8_t commandSize;
251     uint8_t i_string;
252     bool commandStatus;
253
254     /* DEBUG */
255     #ifdef DEBUG_ESP_SEND
256         DEBUG_LED = true;
257     #endif
258
259     /* Default command status */
260     commandStatus = false;
261
262     /* Dont send a command if not IDLE */
263     if(espData.state == ESP_STATE_IDLE)
264     {
265         /* Point to the desired FIFO */
266         p_fifoDesc = &espData.fifoDesc_tx;
267
268         /* Get number of characters to send */
269         commandSize = strlen(p_command);
270
271         /* Check if enough space in FIFO */
272         if(FIFO_GetWriteSpace(p_fifoDesc) >= (commandSize + 2))
273         {
274             /* Loop to add command */
275             for(i_string = 0; i_string < commandSize; i_string++)
276             {
277                 FIFO_Add(p_fifoDesc, (uint8_t) (p_command[i_string]));
278             }
279
280             /* Add CR and LF suffix to FIFO */
281             FIFO_Add(p_fifoDesc, (uint8_t) ('\r'));
282             FIFO_Add(p_fifoDesc, (uint8_t) ('\n'));
283
284             /* Command added to FIFO */
285             espData.transmit = true;
286             commandStatus = true;
287         }
288     }
289     /* DEBUG */
290     #ifdef DEBUG_ESP_SEND
291         DEBUG_LED = false;
292     #endif

```

```

293
294     /* Feedback */
295     return commandStatus;
296 }
297
298 /*****
299
300 /**
301  * @brief _IntHandlerDrvUsartInstance0
302  *
303  * Interrupt instance to manage UART communication to ESP32
304  *
305  */
306 void __ISR(_UART_1_VECTOR, ipl7AUTO) _IntHandlerDrvUsartInstance0(void)
307 {
308     S_Fifo *RX_fifoDescriptor;
309     S_Fifo *TX_fifoDescriptor;
310     uint8_t TX_size;
311     uint8_t TX_BufferFull;
312     uint8_t dataFifo;
313     USART_ERROR usartStatus;
314
315     /* Pointers to fifo descriptors */
316     RX_fifoDescriptor = &espData.fifoDesc_rx;
317     TX_fifoDescriptor = &espData.fifoDesc_tx;
318
319
320     /* DEBUG */
321     #ifdef DEBUG_ESP_UART
322         DEBUG_LED = true;
323     #endif
324
325     /* Reading the error interrupt flag */
326     if(SYS_INT_SourceStatusGet(ESP_INT_SOURCE_USART_ERROR))
327     {
328         /* Clear up the error interrupt flag */
329         SYS_INT_SourceStatusClear(ESP_INT_SOURCE_USART_ERROR);
330     }
331
332     /* Reading the receive interrupt flag */
333     if(SYS_INT_SourceStatusGet(ESP_INT_SOURCE_USART_RECEIVE))
334     {
335         /* Checks overrun or parity error */
336         usartStatus = PLIB_USART_ErrorsGet(ESP_USART_ID);
337
338         if(usartStatus)
339         {
340             /* Errors are auto cleaned when read, except overrun */
341             if ( usartStatus & USART_ERROR_RECEIVER_OVERRUN )
342             {
343                 PLIB_USART_ReceiverOverrunErrorClear(ESP_USART_ID);
344             }
345         }
346         else
347         {
348             espData.receive = true;
349
350             while(PLIB_USART_ReceiverDataIsAvailable(ESP_USART_ID))
351             {
352                 dataFifo = PLIB_USART_ReceiverByteReceive(ESP_USART_ID);
353                 FIFO_Add(RX_fifoDescriptor, dataFifo);
354             }
355
356             /* Clear up the interrupt flag when buffer is empty */
357             SYS_INT_SourceStatusClear(ESP_INT_SOURCE_USART_RECEIVE);
358         }
359     }
360
361     /* Reading the transmit interrupt flag */
362     if(SYS_INT_SourceStatusGet(ESP_INT_SOURCE_USART_TRANSMIT))
363     {
364         TX_size = FIFO_GetReadSpace(TX_fifoDescriptor);
365         TX_BufferFull = PLIB_USART_TransmitterBufferIsFull(ESP_USART_ID);

```

```

366
367     while(TX_size && !TX_BufferFull)
368     {
369         FIFO_GetData(TX_fifoDescriptor, &dataFifo);
370         PLIB_USART_TransmitterByteSend(ESP_USART_ID, dataFifo);
371         TX_size = FIFO_GetReadSpace(TX_fifoDescriptor);
372         TX_BufferFull = PLIB_USART_TransmitterBufferIsFull(ESP_USART_ID);
373     }
374
375     /* Disable the interrupt, to avoid calling ISR continuously*/
376     SYS_INT_SourceDisable(ESP_INT_SOURCE_USART_TRANSMIT);
377
378     /* Clear up the interrupt flag */
379     SYS_INT_SourceStatusClear(ESP_INT_SOURCE_USART_TRANSMIT);
380 }
381
382 /* DEBUG */
383 #ifdef DEBUG_ESP_UART
384     DEBUG_LED = false;
385 #endif
386 }
387
388 /*****
389
390 /* End of File *****/
391

```