```
// UPDATED 2016-03-19


#include "Communications.h"

#include <stdbool.h>
#include <stdlib.h>
#include <xc.h>
#include <p24FV32KA302.h>

#define BUFFER_LENGTH 40  // max size is positive signed character size
#define PORT_COUNT 3 // one based count of the number of ports

//#define RUN_INTERVAL 8000 // run periodically, going too fast causes problems - this is based off a timer, not loops through the program

#define PARAMETER_MAX_COUNT 5
#define PARAMETER_MAX_LENGTH 10

#define CHAR_NULL '\0'
#define COMMAND_SEND_RECEIVE_PRIMER_CHAR '#' // something to run the SPI clock so data can be received
#define COMMAND_START_CHAR '!'
#define COMMAND_END_CHAR '*'
#define COMMAND_DELIMETER ';'

#define RECEIVE_WAIT_COUNT_LIMIT 25
#define RECEIVE_IN_COMMAND_COUNT_LIMIT 253

#define SPI_PORT_0_DIR TRISBbits.TRISB15
#define SPI_PORT_1_DIR TRISBbits.TRISB14
#define SPI_PORT_2_DIR TRISBbits.TRISB12

#define SPI_PORT_0 LATBbits.LATB15
#define SPI_PORT_1 LATBbits.LATB14
#define SPI_PORT_2 LATBbits.LATB12


#define LED1SET LATAbits.LATA2
#define LED1READ PORTAbits.RA2
#define LED1DIR TRISAbits.TRISA2

#define LED2SET LATAbits.LATA3
#define LED2READ PORTAbits.RA3
#define LED2DIR TRISAbits.TRISA3

#define LED3SET LATBbits.LATB4
#define LED3READ PORTBbits.RB4
#define LED3DIR TRISBbits.TRISB4

#define LED4SET LATAbits.LATA4
#define LED4READ PORTAbits.RA4
#define LED4DIR TRISAbits.TRISA4

extern unsigned long tba_powerWatts;
extern unsigned long tba_energyUsedLifetime;

enum receive_status
{
    receive_waiting,
    receive_in_command,
    receive_end_command
};

struct buffer
{
    char data_buffer[ BUFFER_LENGTH + 1];
    unsigned char write_position;
    unsigned char read_position;
```

```
68  };
69
70  extern void delayMS(int);
71
72  bool SPI_receive_data(char *);
73  bool set_current_port(unsigned char *);
74
75  enum receive_status receive_data(struct buffer *, bool *data_received);
76
77  bool process_data(struct buffer *receive_buffer, struct buffer *send_buffer);
78  void process_data_parameterize(char parameters[][PARAMETER_MAX_LENGTH], struct buffer
    *buffer_to_parameterize);
79  bool process_data_parameters(char parameters[][PARAMETER_MAX_LENGTH], struct buffer
    *send_buffer);
80
81  void command_builder1(struct buffer *send_buffer, char* data1);
82  void command_builder2(struct buffer *send_buffer, char* data1, char* data2);
83  void command_builder3(struct buffer *send_buffer, char* data1, char* data2, char* data3);
84  void command_builder4(struct buffer *send_buffer, char* data1, char* data2, char*
    data3, char* data4);
85  void command_builder_add_char(struct buffer *send_buffer, char data);
86  void command_builder_add_string(struct buffer *send_buffer, char *data);
87
88  bool send_data(struct buffer *send_buffer);
89  bool SPI_send_data(char data);
90
91  bool strmatch(char* a, char* b);
92  int strcmp2(char* a, char* b);
93  void strcpy2(char* rcv, char* source);
94
95  void send_end_of_transmission(struct buffer *send_buffer);
96
97  /**********************
98   main code body
99   */
100
101 bool communications()
102 {
103     bool enabledSPI;
104
105     static unsigned char current_port = PORT_COUNT; // port we are on - zero based - 0
        to (PORT_COUNT - 1) we start with max so next port is 0
106     static unsigned char current_port_done = true; // start with true and let normal
        program mechanism automatically init things
107
108     struct buffer send_buffer;
109     static struct buffer receive_buffer;
110
111     static bool end_of_transmission_received = false;
112     bool no_more_to_send; // here to make this more readable
113
114     static enum receive_status receive_current_state;
115     static unsigned int receive_wait_count;
116     static unsigned int receive_in_command_count;
117
118     if (current_port_done == true)
119     {
120         enabledSPI = set_current_port(&current_port);
121
122         if (enabledSPI == true)
123         {
124             current_port_done = false;
125             end_of_transmission_received = false;
126
127             receive_wait_count = 0;
128             receive_in_command_count = 0;
129
130             // put something in the send buffer to run the clock
131             send_buffer.write_position = 0;
```

```
132             send_buffer.read_position = 0;
133             receive_buffer.write_position = 0;
134             receive_buffer.read_position = 0;
135 
136             command_builder_add_char(&send_buffer, COMMAND_SEND_RECEIVE_PRIMER_CHAR);
137 
138         }
139     }
140 
141     bool data_received;
142 
143     receive_current_state = receive_data(&receive_buffer, &data_received);
144     switch (receive_current_state)
145     {
146         case receive_waiting:
147             // count # of times we are waiting for COMMAND_START_CHAR !
148             if (data_received == true)
149             {
150                 receive_wait_count++;
151             }
152 
153             break;
154         case receive_in_command:
155             // count # of times we are in a command
156             // need to check if we somehow missed the COMMAND_END_CHAR *
157             // must be more than max length a command can be
158             if (data_received == true)
159             {
160                 receive_wait_count = 0;
161                 receive_in_command_count++;
162             }
163 
164             break;
165         case receive_end_command:
166 
167             if (process_data(&receive_buffer, &send_buffer) == true)
168             {
169                 end_of_transmission_received = true;
170             }
171             receive_wait_count = 0;
172             receive_in_command_count = 0;
173 
174             break;
175     }
176 
177     no_more_to_send = send_data(&send_buffer);
178 
179     if (no_more_to_send == true)
180     {
181         if (end_of_transmission_received == true)
182         {
183             // make sure trans buffer is empty
184             // the following test is for standard buffer mode only
185             // a different check must be performed if the enhanced buffer is used
186             if (SPI1STATbits.SPITBF == 0b0) // only for standard buffer
187             {
188                 current_port_done = true;
189             }
190         }
191         else if (receive_wait_count >= RECEIVE_WAIT_COUNT_LIMIT)
192         {
193             // not receiving anything valid from slave
194             // just move to the next port - things should clear up on their own
                eventually
195 
196             current_port_done = true;
197         }
198         else if (receive_in_command_count >= RECEIVE_IN_COMMAND_COUNT_LIMIT)
199         {
```

```
200             // received too many characters before the command was ended
201             // likely a garbled COMMAND_END_CHAR or something
202             // just move to the next port - things should clear up on their own
                eventually
203             current_port_done = true;
204         }
205         else
206         {
207             command_builder_add_char(&send_buffer, COMMAND_SEND_RECEIVE_PRIMER_CHAR);
208         }
209     }
210
211     return enabledSPI;
212 }
213
214 bool set_current_port(unsigned char *current_port)
215 {
216     static bool enabledSPI = true;
217
218     if (enabledSPI == true)
219     {
220         SPI1STATbits.SPIEN = 0; //disable master SPI
221         enabledSPI = false;
222
223         SPI_PORT_0 = 1; //disable slave select (1 is disabled)
224         SPI_PORT_1 = 1; //disable slave select (1 is disabled)
225         SPI_PORT_2 = 1; //disable slave select (1 is disabled)
226         //  LED4SET = 0;
227
228     }
229     else
230     {
231         (*current_port)++;
232         if (*current_port >= PORT_COUNT)
233         {
234             *current_port = 0;
235         }
236
237         switch (*current_port)
238         {
239             case 0:
240                 // set correct DO chip select here
241                 SPI_PORT_0 = 0; //enable Slave Select
242                 break;
243             case 1:
244                 // set correct DO the chip select here
245                 SPI_PORT_1 = 0;
246                 //      LED4SET = 1;
247
248                 break;
249             case 2:
250                 // set correct DO the chip select here
251                 SPI_PORT_2 = 0;
252                 break;
253         }
254
255         SPI1STATbits.SPIEN = 1; //enable master SPI
256         enabledSPI = true;
257     }
258
259     return enabledSPI;
260 }
261
262 enum receive_status receive_data(struct buffer *receive_buffer, bool *data_received)
263 {
264     char data;
265
266     static enum receive_status my_status = receive_waiting;
267
```

```
268         if (my_status == receive_end_command)
269         {
270             my_status = receive_waiting;
271         }
272 
273         *data_received = false;
274 
275         if (SPI_receive_data(&data) == true)
276         {
277             *data_received = true;
278 
279 
280             if ((data == COMMAND_START_CHAR) && (my_status != receive_in_command))
281             {
282 
283                 my_status = receive_in_command;
284                 receive_buffer->read_position = 0;
285                 receive_buffer->write_position = 0;
286             }
287 
288             if (my_status == receive_in_command)
289             {
290                 receive_buffer->data_buffer[ receive_buffer->write_position] = data;
291 
292                 receive_buffer->write_position++;
293                 if (receive_buffer->write_position >= BUFFER_LENGTH)
294                 {
295                     receive_buffer->write_position = (BUFFER_LENGTH - 1);
296                 }
297             }
298 
299             if ((my_status == receive_in_command) && (data == COMMAND_END_CHAR))
300             {
301                 my_status = receive_end_command;
302             }
303         }
304 
305         return my_status;
306     }
307 
308     bool process_data(struct buffer *receive_buffer, struct buffer *send_buffer)
309     {
310         bool end_of_transmission_received;
311 
312         // if we are here then the receive buffer must have good data with start and end
            command characters
313         // the characters are not included as they were not added
314 
315         char parameters[PARAMETER_MAX_COUNT][PARAMETER_MAX_LENGTH];
316 
317         process_data_parameterize(parameters, receive_buffer);
318 
319         end_of_transmission_received = process_data_parameters(parameters, send_buffer);
320 
321         return end_of_transmission_received;
322     }
323 
324     void process_data_parameterize(char parameters[][PARAMETER_MAX_LENGTH], struct buffer
        *buffer_to_parameterize)
325     {
326         unsigned char parameter_position = 0;
327         unsigned char parameter_index = 0;
328 
329         // only one command is expected due to the way we read
330         // go through buffer until we hit end char or end of buffer
331 
332         // this is super important - we must initialize the entire array
333         // if we do not we risk passing junk into some functions that assume strings and
            check for NULL
```

```
334         // without NULL a string function could run forever until we die from old age
335         // even then it would keep running
336         for (int inx = 0; inx < PARAMETER_MAX_COUNT; inx++)
337         {
338             parameters[inx][0] = CHAR_NULL;
339         }
340 
341         while ((buffer_to_parameterize->data_buffer[buffer_to_parameterize->read_position ]
            != COMMAND_END_CHAR) && (buffer_to_parameterize->read_position < BUFFER_LENGTH) &&
            (buffer_to_parameterize->read_position != buffer_to_parameterize->write_position))
342         {
343             switch
                (buffer_to_parameterize->data_buffer[buffer_to_parameterize->read_position])
344             {
345                 case COMMAND_START_CHAR:
346                     // this character should never appear
347                     break;
348                 case COMMAND_DELIMETER:
349                     // move to next parameter
350                     parameter_position = 0;
351                     parameter_index++;
352 
353                     if (parameter_index >= PARAMETER_MAX_COUNT)
354                     {
355                         // if we run out of parameters just overwrite the last one
356                         // we should never have this case, but this keeps us from crashing
                            and burning
357                         parameter_index = (PARAMETER_MAX_COUNT - 1);
358                     }
359 
360                     break;
361                 default:
362                     // add the character to the current parameter
363                     parameters[parameter_index][parameter_position] =
                        buffer_to_parameterize->data_buffer[buffer_to_parameterize->read_position
                        ];
364                     parameter_position++;
365                     if (parameter_position >= PARAMETER_MAX_LENGTH)
366                     {
367                         // if our parameter is too long, just overwrite the last character
368                         // we should never have this case, but this keeps us from crashing
                            and burning
369                         parameter_position = (PARAMETER_MAX_LENGTH - 1);
370                     }
371                     parameters[parameter_index][parameter_position] = CHAR_NULL;
372                     break;
373             }
374             buffer_to_parameterize->read_position++;
375         }
376 
377         buffer_to_parameterize->read_position = 0;
378         buffer_to_parameterize->write_position = 0;
379 
380         return;
381     }
382 
383     bool process_data_parameters(char parameters[][PARAMETER_MAX_LENGTH], struct buffer
        *send_buffer)
384     {
385         bool end_of_transmission_received = false;
386 
387         // the 'commands' shown here are for example only
388         // make them whatever is needed
389 
390         // ideally, any new commands are set in a separate function called from one of
            these tests
391         // it's not very clean to call the command builder functions from here
392         // especially if there is some processing to do, like setting a clock or something
393 
```

```
394         if (strmatch(parameters[0], "END") == true)
395         {
396 
397             send_end_of_transmission(send_buffer);
398             end_of_transmission_received = true;
399         }
400         else if (strmatch(parameters[0], "Set") == true)
401         {
402             if (strmatch(parameters[1], "Watts") == true)
403             {
404                 tba_powerWatts = strtoul(parameters[2], NULL, 10);
405                 command_builder2(send_buffer, "Conf", "Watts");
406             }
407             else if (strmatch(parameters[1], "EnUsed") == true)
408             {
409                 // the lifetime energy is currently stored in the command board EEPROM
410                 // power sense at power-up has lifetime energy at 0
411                 // if power sense lifetime energy is < command board lifetime energy we
                    must be in start-up
412                 // send power sense new lifetime energy value
413 
414                 unsigned long tempEnergyUsedLifetime;
415 
416                 tempEnergyUsedLifetime = strtoul(parameters[2], NULL, 10);
417 
418                 if (tempEnergyUsedLifetime < tba_energyUsedLifetime)
419                 {
420                     char temp[12];
421                     //      ultoa( temp, totalUsed, 10 );
422                     ultoa(temp, tba_energyUsedLifetime, 10);
423                     command_builder3(send_buffer, "Set", "EnUsed", temp);
424                 }
425                 else
426                 {
427                     tba_energyUsedLifetime = tempEnergyUsedLifetime;
428                     // done know if we need this here       powerUsed = totalUsed -
                        tba_powerUsedDayStart;
429                     command_builder2(send_buffer, "Conf", "EnUsed");
430                 }
431 
432             }
433                 //  else if( strmatch( parameters[1], "Volts" ) == true )
434                 //  {
435                 //      powerVolts = atoi( parameters[2] );
436                 //      command_builder2( send_buffer, "Conf", "Volts" );
437                 //  }
438                 //  else if( strmatch( parameters[1], "Amps" ) == true )
439                 //  {
440                 //      powerAmps = atoi( parameters[2] );
441                 //      command_builder2( send_buffer, "Conf", "Amps" );
442                 //  }
443             else if (strmatch(parameters[1], "PSVersion") == true)
444             {
445                 command_builder2(send_buffer, "Conf", "PSVersion");
446             }
447 
448             //  else if( strmatch( parameters[1], "LED" ) == true )
449             //  {
450             //      if( strmatch( parameters[2], "On" ) == true )
451             //      {
452             //      command_builder3( send_buffer, "Conf", "LED", "On" );
453             //
454             //      }
455             //      else if( strmatch( parameters[2], "Off" ) == true )
456             //      {
457             //      command_builder3( send_buffer, "Conf", "LED", "Off" );
458             //      }
459             //  }
460             //  else if( strmatch( parameters[1], "LEDB" ) == true )
```

```
461             //  {
462             //      if( strmatch( parameters[2], "On" ) == true )
463             //      {
464             ////         LED1SET = 1;
465             //      command_builder3( send_buffer, "Conf", "LEDB", "On" );
466             //
467             //      }
468             //      else if( strmatch( parameters[2], "Off" ) == true )
469             //      {
470             ////         LED1SET = 0;
471             //      command_builder3( send_buffer, "Conf", "LEDB", "Off" );
472             //      }
473             //  }
474
475
476         }
477         else if (strmatch(parameters[0], "Read") == true)
478         {
479             //  if( strmatch( parameters[1], "LEDB" ) == true )
480             //  {
481             //
482             //      if( LED1READ == 0b1 )
483             //      {
484             //      command_builder3( send_buffer, "Data", "LEDB", "On" );
485             //      }
486             //      else
487             //      {
488             //      command_builder3( send_buffer, "Data", "LEDB", "Off" );
489             //      }
490             //  }
491         }
492
493         // add new parameters as necessary
494         // NEVER check for a higher parameter number than we allocated for.
495         // see earlier comments about NULLS and dying from old age
496
497         return end_of_transmission_received;
498     }
499
500     void command_builder1(struct buffer *send_buffer, char* data1)
501     {
502         command_builder_add_char(send_buffer, COMMAND_SEND_RECEIVE_PRIMER_CHAR);
503         command_builder_add_char(send_buffer, COMMAND_START_CHAR);
504         command_builder_add_string(send_buffer, data1);
505         command_builder_add_char(send_buffer, COMMAND_END_CHAR);
506
507         return;
508     }
509
510     void command_builder2(struct buffer *send_buffer, char* data1, char* data2)
511     {
512         command_builder_add_char(send_buffer, COMMAND_SEND_RECEIVE_PRIMER_CHAR);
513         command_builder_add_char(send_buffer, COMMAND_START_CHAR);
514         command_builder_add_string(send_buffer, data1);
515         command_builder_add_char(send_buffer, COMMAND_DELIMETER);
516         command_builder_add_string(send_buffer, data2);
517         command_builder_add_char(send_buffer, COMMAND_END_CHAR);
518
519         return;
520     }
521
522     void command_builder3(struct buffer *send_buffer, char* data1, char* data2, char* data3)
523     {
524         command_builder_add_char(send_buffer, COMMAND_SEND_RECEIVE_PRIMER_CHAR);
525         command_builder_add_char(send_buffer, COMMAND_START_CHAR);
526         command_builder_add_string(send_buffer, data1);
527         command_builder_add_char(send_buffer, COMMAND_DELIMETER);
528         command_builder_add_string(send_buffer, data2);
529         command_builder_add_char(send_buffer, COMMAND_DELIMETER);
```

```
530         command_builder_add_string(send_buffer, data3);
531         command_builder_add_char(send_buffer, COMMAND_END_CHAR);
532 
533         return;
534     }
535 
536     void command_builder4(struct buffer *send_buffer, char* data1, char* data2, char*
    data3, char* data4)
537     {
538         command_builder_add_char(send_buffer, COMMAND_SEND_RECEIVE_PRIMER_CHAR);
539         command_builder_add_char(send_buffer, COMMAND_START_CHAR);
540         command_builder_add_string(send_buffer, data1);
541         command_builder_add_char(send_buffer, COMMAND_DELIMETER);
542         command_builder_add_string(send_buffer, data2);
543         command_builder_add_char(send_buffer, COMMAND_DELIMETER);
544         command_builder_add_string(send_buffer, data3);
545         command_builder_add_char(send_buffer, COMMAND_DELIMETER);
546         command_builder_add_string(send_buffer, data4);
547         command_builder_add_char(send_buffer, COMMAND_END_CHAR);
548 
549         return;
550     }
551 
552     void command_builder_add_char(struct buffer *send_buffer, char data)
553     {
554         send_buffer->data_buffer[send_buffer->write_position] = data;
555 
556         send_buffer->write_position++;
557         if (send_buffer->write_position >= BUFFER_LENGTH)
558         {
559             send_buffer->write_position = 0;
560         }
561 
562         return;
563     }
564 
565     void command_builder_add_string(struct buffer *send_buffer, char *data_string)
566     {
567         for (int inx = 0; data_string[inx] != CHAR_NULL; inx++)
568         {
569             command_builder_add_char(send_buffer, data_string[inx]);
570         }
571 
572         return;
573     }
574 
575     bool send_data(struct buffer *send_buffer)
576     {
577         bool send_end;
578 
579         if (send_buffer->read_position == send_buffer->write_position)
580         {
581             send_end = true;
582         }
583         else
584         {
585             send_end = false;
586             if (SPI_send_data(send_buffer->data_buffer[send_buffer->read_position]) == true)
587             {
588                 send_buffer->read_position++;
589                 if (send_buffer->read_position >= BUFFER_LENGTH)
590                 {
591                     send_buffer->read_position = 0;
592                 }
593             }
594         }
595 
596         return send_end;
597     }
```

```
598
599 bool strmatch(char* a, char* b)
600 {
601     int result;
602     bool match;
603
604     static int co = 0;
605     co++;
606
607     result = strcmp2(a, b);
608
609     if (result == 0)
610     {
611         match = true;
612     }
613     else
614     {
615         match = false;
616     }
617
618     return match;
619 }
620
621 int strcmp2(char* a, char* b)
622 {
623     int inx = 0;
624     int match = 0;
625
626     while ((a[inx] != CHAR_NULL) && (b[inx] != CHAR_NULL) && (match == 0))
627     {
628         if (a[inx] > b[inx])
629         {
630             match = 1;
631         }
632         else if (a[inx] < b[inx])
633         {
634             match = -1;
635         }
636         else if (a[inx] == b[inx])
637         {
638             //do nothing
639         }
640
641         inx++;
642     }
643
644     if ((a[inx] == CHAR_NULL) && (b[inx] != CHAR_NULL))
645     {
646         match = -1;
647     }
648     else if ((a[inx] != CHAR_NULL) && (b[inx] == CHAR_NULL))
649     {
650         match = 1;
651     }
652
653
654     return match;
655 }
656
657 bool SPI_receive_data(char *data)
658 {
659     bool recvGood = false;
660
661     if (SPI1STATbits.SPIRBF == 1)
662     {
663         *data = SPI1BUF;
664         recvGood = true;
665     }
666
```

```
667         return recvGood;
668     }
669
670     bool SPI_send_data(char data)
671     {
672         bool sendGood = false;
673
674         if (SPI1STATbits.SPITBF == 0) //if in enhance mode use SPI1STATbits.SR1MPT
675         {
676             SPI1BUF = data;
677             sendGood = true;
678         }
679
680         return sendGood;
681     }
682
683     /************************/
684     // RESPONSES
685
686     void send_end_of_transmission(struct buffer *send_buffer)
687     {
688         command_builder1(send_buffer, "END");
689
690         return;
691     }
692
693     void SPIMasterInit(void)
694     {
695         static bool firstRun = true;
696
697         // make sure analog is turned off - it messes with the pins
698         ANSA = 0;
699         ANSB = 0;
700
701         TRISBbits.TRISB10 = 0b1; // SDI1
702         TRISBbits.TRISB11 = 0b1; // SCK1 (seems this should be set output, but 0b0 does not
        work)
703         TRISBbits.TRISB13 = 0b0; // SDO1
704
705         SPI_PORT_0_DIR = 0;
706         SPI_PORT_1_DIR = 0;
707         SPI_PORT_2_DIR = 0;
708
709         SPI_PORT_0 = 1;
710         SPI_PORT_1 = 1;
711         SPI_PORT_2 = 1;
712
713         //SPI1 Initialize
714         SPI1CON1bits.MSTEN = 1; //making SPI1 Master
715
716         SPI1CON1bits.DISSCK = 0b0; // SPI clock enabled
717         SPI1CON1bits.DISSDO = 0b0; // SDO used
718         SPI1CON1bits.MODE16 = 0b0; // 8 bit mode
719         SPI1CON1bits.SMP = 0b1; // sample phase mode end
720         SPI1CON1bits.CKE = 0b1; // serial data changes on active to idle clock state
721         SPI1CON1bits.SSEN = 0b0; // not a slave
722         SPI1CON1bits.CKP = 0b1; // clock idle is high
723         SPI1CON1bits.SPRE = 0b110; // secondary prescale 8:1
724         SPI1CON1bits.PPRE = 0b11; // primary prescale 1:1
725         //    SPI1CON1bits.PPRE = 0b00; // primary prescale 1:1
726
727         SPI1CON2bits.FRMEN = 0b0; // frame mode, unused
728         SPI1CON2bits.SPIFSD = 0b0; // frame mode, unused
729         SPI1CON2bits.SPIFPOL = 0b0; // frame mode, unused
730         SPI1CON2bits.SPIFE = 0b0; // frame mode, unused
731
732         SPI1CON2bits.SPIBEN = 0b0; // 1=enhanced buffer mode
733
734         SPI1STATbits.SPIROV = 0; //clear flag for overflow data
```

```
735
736
737         // do not use the interrupt , could not get it to work
738         //    SPI1STATbits.SISEL = 0b001;
739         //
740         //    IFS0bits.SPI1IF = 0;
741         //    IEC0bits.SPI1IE = 1;
742
743
744
745
746         //SPI1BUF = SPI1BUF;
747         //    SPI1STATbits.SPIEN = 1; //enable SPI
748
749         if (firstRun == true)
750         {
751             // set timer up here
752             T1CONbits.TSIDL = 0b1; //Discontinue module operation when device enters idle
                mode
753             T1CONbits.T1ECS = 0b00; // Timer1 uses Secondary Oscillator (SOSC) as the clock
                soource)
754             T1CONbits.TGATE = 0b0; // Gated time accumulation is disabled
755             T1CONbits.TSYNC = 0b0; // Do not synchronize external clock input (asynchronous)
756             T1CONbits.TCS = 0b0; //use internal clock
757
758
759             T1CONbits.TCKPS = 0b00; // Timer 1 Input Clock Prescale (11-256)(10-64) (01-8)
                (00-1)
760             TMR1 = 0x0000; // start timer at 0
761
762             T1CONbits.TON = 0b1; //turn on timer
763             firstRun = false;
764         }
765         return;
766     }
767
768     //void __attribute__((__interrupt__,__auto_psv__)) _SPI1Interrupt(void)
769     //{
770     //    // we received a byte
771     //    IFS0bits.SPI1IF = 0;
772     //
773     //    rcvSPI = true;
774     //    rcvChar =  SPI1BUF;
775     //
776     //    return;
777     //
778     //}
779
```