```c
#include <xc.h>
//#include <p18cxxx.h>
#include <p18f25k22.h>


#include "config.h"
#include "Communications.h"
#include <stdbool.h>
#include <stdlib.h>

//#include <pic18f25k22.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 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 LEDSET LATBbits.LATB4
//#define LEDDIR TRISBbits.RB4
//#define LEDREAD PORTBbits.RB4

bool SPI_transmit_wait;

enum receive_status
{
    receive_waiting,
    receive_in_command,
    receive_end_command
};

struct buffer
{
    char data_buffer[ BUFFER_LENGTH];
    unsigned char write_position;
    unsigned char read_position;
};

extern unsigned long meterWatts;
extern unsigned long meterEnergyUsed;



bool SPI_receive_data(char* data);
void set_current_port(unsigned char *);
enum receive_status receive_data(struct buffer *);
bool process_data(struct buffer *receive_buffer, struct buffer *send_buffer);
void process_data_parameterize(char parameters[PARAMETER_MAX_COUNT][PARAMETER_MAX_LENGTH], struct buffer *buffer_to_parameterize);
bool process_data_parameters(char parameters[PARAMETER_MAX_COUNT][PARAMETER_MAX_LENGTH], struct buffer *send_buffer);

void command_builder1(struct buffer *send_buffer, char* data1);
void command_builder2(struct buffer *send_buffer, char* data1, char* data2);
void command_builder3(struct buffer *send_buffer, char* data1, char* data2, char* data3);
void command_builder4(struct buffer *send_buffer, char* data1, char* data2, char* data3, char* data4);
void command_builder_add_char(struct buffer *send_buffer, char data);
void command_builder_add_string(struct buffer *send_buffer, char *data);

```

```
65  bool send_data(struct buffer *send_buffer);
66  bool SPI_send_data(char data);
67
68  bool strmatch(char* a, char* b);
69  int strcmp2(char* a, char* b);
70  void strcpy2(char* rcv, char* source);
71
72  void resetCommunications(struct buffer * receive_buffer);
73  void SPISlaveInit(void);
74
75
76  void send_end_of_transmission(struct buffer *send_buffer);
77  void com_command_testLED(struct buffer * send_buffer);
78  void com_command_setPower(struct buffer * send_buffer);
79  void com_command_setEnergyUsed(struct buffer * send_buffer);
80  void com_command_setVolts(struct buffer * send_buffer);
81  void com_command_setAmps(struct buffer * send_buffer);
82  void com_command_readCalibration(struct buffer * send_buffer);
83  void com_command_setVersion(struct buffer * send_buffer);
84
85  /**********************
86   main code body
87   */
88
89  void communications(bool firstTime)
90  {
91      static struct buffer receive_buffer;
92      static struct buffer send_buffer;
93
94      static bool end_of_transmission_received = false;
95      bool no_more_to_send; // here to make this more readable
96
97      static enum receive_status receive_current_state;
98
99
100     if (firstTime == true)
101     {
102         SPISlaveInit();
103         send_buffer.write_position = 0;
104         send_buffer.read_position = 0;
105         resetCommunications(&send_buffer);
106     }
107     else
108     {
109         receive_current_state = receive_data(&receive_buffer);
110
111         switch (receive_current_state)
112         {
113             case receive_waiting:
114                 // don't need to worry about it too much
115                 break;
116             case receive_in_command:
117                 // don't need to worry about it too much
118                 break;
119             case receive_end_command:
120
121                 if (process_data(&receive_buffer, &send_buffer) == true)
122                 {
123                     end_of_transmission_received = true;
124                 }
125
126                 break;
127         }
128
129         no_more_to_send = send_data(&send_buffer);
130
131
132         static bool last_state_active = false;
133         if (PORTBbits.SS2 == 0b1)
```

```
134         {
135             last_state_active = false;
136         }
137         else
138         {
139             if (last_state_active == false)
140             {
141                 resetCommunications(&send_buffer);
142             }
143 
144             last_state_active = true;
145         }
146 
147     }
148 
149     return;
150 }
151 
152 void resetCommunications(struct buffer * send_buffer)
153 {
154 
155     static int commState = 0;
156 
157 
158     SSP2CON1bits.SSPEN = 0; //disable SPI
159     __delay_ms(1);
160     SSP2CON1bits.SSPEN = 1; //enable SPI
161 
162     SSP2CON1bits.WCOL = 0;
163     SPI_transmit_wait = false;
164 
165     send_buffer->read_position = 0;
166     send_buffer->write_position = 0;
167 
168 
169     // set up command state machine
170     // do we repeat a command if we did not hit END command?
171     commState++;
172     switch (commState)
173     {
174         case 1:
175             com_command_setVersion(send_buffer);
176             break;
177         case 2:
178             com_command_setPower(send_buffer);
179             break;
180         case 3:
181             com_command_setEnergyUsed(send_buffer);
182             //  break;
183             //    case 4:
184             //  com_command_setAmps( send_buffer );
185             //  break;
186             //    case 5:
187             //  com_command_readCalibration( send_buffer );
188             //  break;
189             //    case 6:
190             //  com_command_testLED( send_buffer );
191             //  break;
192         default:
193             commState = 0;
194             break;
195     }
196     return;
197 }
198 
199 enum receive_status receive_data(struct buffer * receive_buffer)
200 {
201     char data;
202 
```

```
203         static enum receive_status my_status = receive_waiting;
204
205         if (my_status == receive_end_command)
206         {
207             my_status = receive_waiting;
208         }
209
210         if (SPI_receive_data(&data) == true)
211         {
212             if ((data == COMMAND_START_CHAR) && (my_status != receive_in_command))
213             {
214                 my_status = receive_in_command;
215                 receive_buffer->read_position = 0;
216                 receive_buffer->write_position = 0;
217             }
218
219             if (my_status == receive_in_command)
220             {
221                 receive_buffer->data_buffer[ receive_buffer->write_position] = data;
222
223                 receive_buffer->write_position++;
224                 if (receive_buffer->write_position >= BUFFER_LENGTH)
225                 {
226                     receive_buffer->write_position = (BUFFER_LENGTH - 1);
227                 }
228             }
229
230             if ((my_status == receive_in_command) && (data == COMMAND_END_CHAR))
231             {
232                 my_status = receive_end_command;
233             }
234         }
235
236         return my_status;
237     }
238
239     bool process_data(struct buffer *receive_buffer, struct buffer * send_buffer)
240     {
241         bool end_of_transmission_received;
242
243         // if we are here then the receive buffer must have good data with start and end
            command characters
244         // the characters are not included as they were stripped from the incoming data
245
246         char parameters[PARAMETER_MAX_COUNT][PARAMETER_MAX_LENGTH];
247
248         process_data_parameterize(parameters, receive_buffer);
249
250         end_of_transmission_received = process_data_parameters(parameters, send_buffer);
251
252         return end_of_transmission_received;
253
254     }
255
256     void process_data_parameterize(char
        parameters[PARAMETER_MAX_COUNT][PARAMETER_MAX_LENGTH], struct buffer *
        buffer_to_parameterize)
257     {
258         unsigned char parameter_position = 0;
259         unsigned char parameter_index = 0;
260
261         // only one command is expected due to the way we read
262         // go through buffer until we hit end char or end of buffer
263
264         // this is super important - we must initialize the entire array
265         // if we do not we risk passing junk into some functions that assume strings and
            check for NULL
266         // without NULL a string function could run forever until we die from old age
267         // even then it would keep running
```

```
268         for (int inx = 0; inx < PARAMETER_MAX_COUNT; inx++)
269         {
270             parameters[inx][0] = CHAR_NULL;
271         }
272 
273         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))
274         {
275             switch
                (buffer_to_parameterize->data_buffer[buffer_to_parameterize->read_position])
276             {
277                 case COMMAND_START_CHAR:
278                     // this character should never appear
279                     break;
280                 case COMMAND_DELIMETER:
281                     // move to next parameter
282                     parameter_position = 0;
283                     parameter_index++;
284 
285                     if (parameter_index >= PARAMETER_MAX_COUNT)
286                     {
287                         // if we run out of parameters just overwrite the last one
288                         // we should never have this case, but this keeps us from crashing
                            and burning
289                         parameter_index = (PARAMETER_MAX_COUNT - 1);
290                     }
291 
292                     break;
293                 default:
294                     // add the character to the current parameter
295                     parameters[parameter_index][parameter_position] =
                        buffer_to_parameterize->data_buffer[buffer_to_parameterize->read_position
                        ];
296                     parameter_position++;
297                     if (parameter_position >= PARAMETER_MAX_LENGTH)
298                     {
299                         // if our parameter is too long, just overwrite the last character
300                         // we should never have this case, but this keeps us from crashing
                            and burning
301                         parameter_position = (PARAMETER_MAX_LENGTH - 1);
302                     }
303 
304                     // always make the last character a null
305                     parameters[parameter_index][parameter_position] = CHAR_NULL;
306                     break;
307             }
308 
309             buffer_to_parameterize->read_position++;
310         }
311 
312         return;
313     }
314 
315     bool process_data_parameters(char
    parameters[PARAMETER_MAX_COUNT][PARAMETER_MAX_LENGTH], struct buffer * send_buffer)
316     {
317         bool end_of_transmission_received = false;
318 
319 
320         if (strmatch(parameters[0], "END") == true)
321         {
322             //  if( LEDSET == 1 )
323             //  {
324             //      LEDSET = 0;
325             //  }
326             //  else
327             //  {
328             //      LEDSET = 1;
```

```
329             //  }
330 
331             end_of_transmission_received = true;
332         }
333         else if (strmatch(parameters[0], "Set") == true)
334         {
335             if (strmatch(parameters[1], "Calibration") == true)
336             {
337                 // set the calibration value for the current sense, if required
338             }
339             else if (strmatch(parameters[1], "EnUsed") == true)
340             {
341                 // set the Energy used
342                 // this likely means that the command board had a stored power used greater
                    than we have here.
343                 // this happens when the power is lost - current sense starts at 0, command
                    board stores in EEPROM
344 
345                 meterEnergyUsed = atol(parameters[2]);
346                 com_command_setEnergyUsed(send_buffer);
347             }
348 
349 
350             //meterEnergyUsed
351 
352         }
353         else if (strmatch(parameters[0], "Read") == true)
354         {
355             // nothing to read right now
356         }
357         else if (strmatch(parameters[0], "Data") == true)
358         {
359             if (strmatch(parameters[1], "LEDB") == true)
360             {
361                 if (strmatch(parameters[2], "On") == true)
362                 {
363                     command_builder3(send_buffer, "Set", "LEDB", "Off");
364                 }
365                 else if (strmatch(parameters[2], "Off") == true)
366                 {
367                     command_builder3(send_buffer, "Set", "LEDB", "On");
368                 }
369             }
370         }
371         else if (strmatch(parameters[0], "Conf") == true)
372         {
373             if (strmatch(parameters[1], "LEDB") == true)
374             {
375                 send_end_of_transmission(send_buffer);
376             }
377             else if (strmatch(parameters[1], "Watts") == true)
378             {
379                 send_end_of_transmission(send_buffer);
380             }
381             else if (strmatch(parameters[1], "EnUsed") == true)
382             {
383                 send_end_of_transmission(send_buffer);
384             }
385             else if (strmatch(parameters[1], "Volts") == true)
386             {
387                 send_end_of_transmission(send_buffer);
388             }
389             else if (strmatch(parameters[1], "Amps") == true)
390             {
391                 send_end_of_transmission(send_buffer);
392             }
393             else if (strmatch(parameters[1], "PSVersion") == true)
394             {
395                 send_end_of_transmission(send_buffer);
```

```
396             }
397         }
398
399         // add new parameters as necessary
400         // NEVER check for a higher parameter number than we allocated for.
401         // see earlier comments about NULLS and dying from old age
402
403         return end_of_transmission_received;
404     }
405
406     void command_builder1(struct buffer *send_buffer, char* data1)
407     {
408         command_builder_add_char(send_buffer, COMMAND_SEND_RECEIVE_PRIMER_CHAR);
409         command_builder_add_char(send_buffer, COMMAND_SEND_RECEIVE_PRIMER_CHAR);
410         command_builder_add_char(send_buffer, COMMAND_START_CHAR);
411         command_builder_add_string(send_buffer, data1);
412         command_builder_add_char(send_buffer, COMMAND_END_CHAR);
413
414         return;
415     }
416
417     void command_builder2(struct buffer *send_buffer, char* data1, char* data2)
418     {
419         command_builder_add_char(send_buffer, COMMAND_SEND_RECEIVE_PRIMER_CHAR);
420         command_builder_add_char(send_buffer, COMMAND_SEND_RECEIVE_PRIMER_CHAR);
421         command_builder_add_char(send_buffer, COMMAND_START_CHAR);
422         command_builder_add_string(send_buffer, data1);
423         command_builder_add_char(send_buffer, COMMAND_DELIMETER);
424         command_builder_add_string(send_buffer, data2);
425         command_builder_add_char(send_buffer, COMMAND_END_CHAR);
426
427         return;
428     }
429
430     void command_builder3(struct buffer *send_buffer, char* data1, char* data2, char* data3)
431     {
432         command_builder_add_char(send_buffer, COMMAND_SEND_RECEIVE_PRIMER_CHAR);
433         command_builder_add_char(send_buffer, COMMAND_SEND_RECEIVE_PRIMER_CHAR);
434         command_builder_add_char(send_buffer, COMMAND_START_CHAR);
435         command_builder_add_string(send_buffer, data1);
436         command_builder_add_char(send_buffer, COMMAND_DELIMETER);
437         command_builder_add_string(send_buffer, data2);
438         command_builder_add_char(send_buffer, COMMAND_DELIMETER);
439         command_builder_add_string(send_buffer, data3);
440         command_builder_add_char(send_buffer, COMMAND_END_CHAR);
441
442         return;
443     }
444
445     void command_builder4(struct buffer *send_buffer, char* data1, char* data2, char*
    data3, char* data4)
446     {
447         command_builder_add_char(send_buffer, COMMAND_SEND_RECEIVE_PRIMER_CHAR);
448         command_builder_add_char(send_buffer, COMMAND_SEND_RECEIVE_PRIMER_CHAR);
449         command_builder_add_char(send_buffer, COMMAND_START_CHAR);
450         command_builder_add_string(send_buffer, data1);
451         command_builder_add_char(send_buffer, COMMAND_DELIMETER);
452         command_builder_add_string(send_buffer, data2);
453         command_builder_add_char(send_buffer, COMMAND_DELIMETER);
454         command_builder_add_string(send_buffer, data3);
455         command_builder_add_char(send_buffer, COMMAND_DELIMETER);
456         command_builder_add_string(send_buffer, data4);
457         command_builder_add_char(send_buffer, COMMAND_END_CHAR);
458
459         return;
460     }
461
462     void command_builder_add_char(struct buffer *send_buffer, char data)
463     {
```

```
464         send_buffer->data_buffer[send_buffer->write_position] = data;
465 
466         send_buffer->write_position++;
467         if (send_buffer->write_position >= BUFFER_LENGTH)
468         {
469             send_buffer->write_position = 0;
470         }
471 
472         return;
473     }
474 
475     void command_builder_add_string(struct buffer *send_buffer, char *data_string)
476     {
477         for (int inx = 0; data_string[inx] != CHAR_NULL; inx++)
478         {
479             command_builder_add_char(send_buffer, data_string[inx]);
480         }
481 
482         return;
483     }
484 
485     bool send_data(struct buffer * send_buffer)
486     {
487         bool send_end;
488 
489         if (send_buffer->read_position == send_buffer->write_position)
490         {
491             send_end = true;
492             SPI_send_data('\0');
493         }
494         else
495         {
496             send_end = false;
497 
498 
499             if (SPI_send_data(send_buffer->data_buffer[send_buffer->read_position]) == true)
500             {
501 
502                 send_buffer->read_position++;
503                 if (send_buffer->read_position >= BUFFER_LENGTH)
504                 {
505                     send_buffer->read_position = 0;
506                 }
507             }
508         }
509 
510         return send_end;
511     }
512 
513     bool strmatch(char* a, char* b)
514     {
515         int result;
516         bool match;
517 
518         result = strcmp2(a, b);
519 
520         match = (result == 0) ? true : false;
521 
522         return match;
523     }
524 
525     int strcmp2(char* a, char* b)
526     {
527         int inx = 0;
528         int match = 0;
529 
530         while ((a[inx] != CHAR_NULL) && (b[inx] != CHAR_NULL) && (match == 0))
531         {
532             if (a[inx] > b[inx])
```

```
533             {
534                 match = 1;
535             }
536             else if (a[inx] < b[inx])
537             {
538                 match = -1;
539             }
540             else if (a[inx] == b[inx])
541             {
542                 //do nothing = never reset to zero
543             }
544 
545             inx++;
546         }
547 
548 
549         if ((a[inx] == CHAR_NULL) && (b[inx] != CHAR_NULL))
550         {
551             match = -1;
552         }
553         else if ((a[inx] != CHAR_NULL) && (b[inx] == CHAR_NULL))
554         {
555             match = 1;
556         }
557 
558         return match;
559 
560     }
561 
562     bool SPI_receive_data(char *data)
563     {
564 
565         bool recvGood = false;
566 
567         if (SSP2STATbits.BF == 1)
568         {
569             *data = SSP2BUF;
570             recvGood = true;
571             SSP2CON1bits.WCOL = 0;
572             SPI_transmit_wait = false;
573         }
574         else
575         {
576             recvGood = false;
577         }
578 
579         return recvGood;
580 
581     }
582 
583     bool SPI_send_data(char data)
584     {
585         bool sendGood = false;
586 
587         if (SPI_transmit_wait == false)
588         {
589             SSP2BUF = data;
590             SPI_transmit_wait = true;
591             sendGood = true;
592         }
593         else
594         {
595             sendGood = false;
596         }
597 
598         return sendGood;
599     }
600 
601     /***********************/
```

```
602  // RESPONSES
603
604  void send_end_of_transmission(struct buffer * send_buffer)
605  {
606      command_builder1(send_buffer, "END");
607
608      return;
609  }
610
611  void com_command_testLED(struct buffer * send_buffer)
612  {
613      command_builder2(send_buffer, "Read", "LEDB");
614
615      return;
616  }
617
618  void com_command_setPower(struct buffer * send_buffer)
619  {
620
621      char temp[12];
622
623      ultoa(temp, meterWatts, 10);
624      command_builder3(send_buffer, "Set", "Watts", temp);
625
626      return;
627  }
628
629  void com_command_setEnergyUsed(struct buffer * send_buffer)
630  {
631      char temp[12];
632
633      ultoa(temp, meterEnergyUsed, 10);
634
635      command_builder3(send_buffer, "Set", "EnUsed", temp);
636
637      return;
638
639
640  }
641
642  void com_command_setVolts(struct buffer * send_buffer)
643  {
644      command_builder3(send_buffer, "Set", "Volts", "222");
645
646      return;
647  }
648
649  void com_command_setAmps(struct buffer * send_buffer)
650  {
651      command_builder3(send_buffer, "Set", "Amps", "333");
652
653      return;
654  }
655
656  void com_command_readCalibration(struct buffer * send_buffer)
657  {
658      command_builder2(send_buffer, "Read", "Calibration");
659
660      return;
661  }
662
663  void com_command_setVersion(struct buffer * send_buffer)
664  {
665      command_builder3(send_buffer, "Set", "PSVersion", "444");
666
667  }
668
669  void SPISlaveInit(void)
670  {
```

```
671
672         TRISAbits.TRISA0 = 0; // pin 2 connected as an output for pulse
673         TRISAbits.TRISA1 = 1; // pin 3 connected as an input for pulse
674         //    LEDDIR = 0; // pin 25 connected as an output for LED
675         TRISCbits.TRISC3 = 0; // pin 14 connected as an output for pulse freq.
676         TRISCbits.TRISC5 = 0; // pin 16 connected as an output for pulse freq.
677         TRISCbits.TRISC6 = 0; // set pin 17 as an output for MCLR
678         TRISCbits.TRISC7 = 0; // set pin 18 as an output for pulse freq.
679         ANSELAbits.ANSA1 = 0b0; // turn off analog to digital conversion
680
681         LATCbits.LATC6 = 1; // set the MCLR of the MCP high
682         LATCbits.LATC3 = 1; // set pin 14 to a 1 to set freq. control F2 for pulse
683         LATCbits.LATC5 = 1; // set pin 16 to a 1 to set freq. control F1 for pulse
684         LATCbits.LATC7 = 1; // set pin 18 to a 1 to set freq. control F0 for pulse
685
686
687         SSP2CON1bits.SSPEN = 0; //Synchronous Serial Port Enable bit
688
689         ANSELBbits.ANSB0 = 0b0;
690         ANSELBbits.ANSB1 = 0b0;
691         ANSELBbits.ANSB2 = 0b0;
692         ANSELBbits.ANSB3 = 0b0;
693
694         TRISBbits.RB0 = 0b1;
695         TRISBbits.RB1 = 0b1;
696         TRISBbits.RB2 = 0b1;
697         TRISBbits.RB3 = 0b0;
698
699         SSP2STATbits.SMP = 0;
700         SSP2STATbits.CKE = 1;
701
702         SSP2CON1bits.WCOL = 0; //Write Collision Detect bit
703         SSP2CON1bits.SSPOV = 0; //Receive Overflow Indicator bit
704         SSP2CON1bits.SSPEN = 0; //Synchronous Serial Port Enable bit
705         SSP2CON1bits.CKP = 1; //Clock Polarity Select bit
706         SSP2CON1bits.SSPM = 0b0100; //Synchronous Serial Port Mode Select bits
707
708
709         SSP2CON3 = 0x00;
710         SSP2CON3bits.BOEN = 0b0; //Buffer Overwrite Enable bit
711
712         SSP2CON1bits.SSPEN = 1; //Synchronous Serial Port Enable bit
713
714         //    SPIWatchdogTimerInit();
715
716         return;
717     }
718
```