```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);

```

```
bool send_data(struct buffer *send_buffer);
bool SPI_send_data(char data);

bool strmatch(char* a, char* b);
int strcmp2(char* a, char* b);
void strcpy2(char* rcv, char* source);

void resetCommunications(struct buffer * receive_buffer);
void SPISlaveInit(void);


void send_end_of_transmission(struct buffer *send_buffer);
void com_command_testLED(struct buffer * send_buffer);
void com_command_setPower(struct buffer * send_buffer);
void com_command_setEnergyUsed(struct buffer * send_buffer);
void com_command_setVolts(struct buffer * send_buffer);
void com_command_setAmps(struct buffer * send_buffer);
void com_command_readCalibration(struct buffer * send_buffer);
void com_command_setVersion(struct buffer * send_buffer);

/***********************
 main code body
 */

void communications(bool firstTime)
{
    static struct buffer receive_buffer;
    static struct buffer send_buffer;

    static bool end_of_transmission_received = false;
    bool no_more_to_send; // here to make this more readable

    static enum receive_status receive_current_state;


    if (firstTime == true)
    {
        SPISlaveInit();
        send_buffer.write_position = 0;
        send_buffer.read_position = 0;
        resetCommunications(&send_buffer);
    }
    else
    {
        receive_current_state = receive_data(&receive_buffer);

        switch (receive_current_state)
        {
            case receive_waiting:
                // don't need to worry about it too much
                break;
            case receive_in_command:
                // don't need to worry about it too much
                break;
            case receive_end_command:

                if (process_data(&receive_buffer, &send_buffer) == true)
                {
                    end_of_transmission_received = true;
                }

                break;
        }

        no_more_to_send = send_data(&send_buffer);


        static bool last_state_active = false;
        if (PORTBbits.SS2 == 0b1)
        {
            last_state_active = false;
        }
        else
        {
            if (last_state_active == false)
            {
                resetCommunications(&send_buffer);
            }

            last_state_active = true;
        }

    }

    return;
}

void resetCommunications(struct buffer * send_buffer)
{

    static int commState = 0;


    SSP2CON1bits.SSPEN = 0; //disable SPI
    __delay_ms(1);
    SSP2CON1bits.SSPEN = 1; //enable SPI

    SSP2CON1bits.WCOL = 0;
    SPI_transmit_wait = false;

    send_buffer->read_position = 0;
    send_buffer->write_position = 0;


    // set up command state machine
    // do we repeat a command if we did not hit END command?
    commState++;
    switch (commState)
    {
        case 1:
            com_command_setVersion(send_buffer);
            break;
        case 2:
            com_command_setPower(send_buffer);
            break;
        case 3:
            com_command_setEnergyUsed(send_buffer);
            //  break;
            //    case 4:
            //  com_command_setAmps( send_buffer );
            //  break;
            //    case 5:
            //  com_command_readCalibration( send_buffer );
            //  break;
            //    case 6:
            //  com_command_testLED( send_buffer );
            //  break;
        default:
            commState = 0;
            break;
    }
    return;
}

enum receive_status receive_data(struct buffer * receive_buffer)
{
    char data;

    static enum receive_status my_status = receive_waiting;

    if (my_status == receive_end_command)
    {
        my_status = receive_waiting;
    }

    if (SPI_receive_data(&data) == true)
    {
        if ((data == COMMAND_START_CHAR) && (my_status != receive_in_command))
        {
            my_status = receive_in_command;
            receive_buffer->read_position = 0;
            receive_buffer->write_position = 0;
        }

        if (my_status == receive_in_command)
        {
            receive_buffer->data_buffer[ receive_buffer->write_position] = data;

            receive_buffer->write_position++;
            if (receive_buffer->write_position >= BUFFER_LENGTH)
            {
                receive_buffer->write_position = (BUFFER_LENGTH - 1);
            }
        }

        if ((my_status == receive_in_command) && (data == COMMAND_END_CHAR))
        {
            my_status = receive_end_command;
        }
    }

    return my_status;
}

bool process_data(struct buffer *receive_buffer, struct buffer * send_buffer)
{
    bool end_of_transmission_received;

    // if we are here then the receive buffer must have good data with start and end command characters
    // the characters are not included as they were stripped from the incoming data

    char parameters[PARAMETER_MAX_COUNT][PARAMETER_MAX_LENGTH];

    process_data_parameterize(parameters, receive_buffer);

    end_of_transmission_received = process_data_parameters(parameters, send_buffer);

    return end_of_transmission_received;

}

void process_data_parameterize(char parameters[PARAMETER_MAX_COUNT][PARAMETER_MAX_LENGTH], struct buffer * buffer_to_parameterize)
{
    unsigned char parameter_position = 0;
    unsigned char parameter_index = 0;

    // only one command is expected due to the way we read
    // go through buffer until we hit end char or end of buffer

    // this is super important - we must initialize the entire array
    // if we do not we risk passing junk into some functions that assume strings and check for NULL
    // without NULL a string function could run forever until we die from old age
    // even then it would keep running
    for (int inx = 0; inx < PARAMETER_MAX_COUNT; inx++)
    {
        parameters[inx][0] = CHAR_NULL;
    }

    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))
    {
        switch
        (buffer_to_parameterize->data_buffer[buffer_to_parameterize->read_position])
        {
            case COMMAND_START_CHAR:
                // this character should never appear
                break;
            case COMMAND_DELIMETER:
                // move to next parameter
                parameter_position = 0;
                parameter_index++;

                if (parameter_index >= PARAMETER_MAX_COUNT)
                {
                    // if we run out of parameters just overwrite the last one
                    // we should never have this case, but this keeps us from crashing
                    and burning
                    parameter_index = (PARAMETER_MAX_COUNT - 1);
                }

                break;
            default:
                // add the character to the current parameter
                parameters[parameter_index][parameter_position] =
                buffer_to_parameterize->data_buffer[buffer_to_parameterize->read_position
                ];
                parameter_position++;
                if (parameter_position >= PARAMETER_MAX_LENGTH)
                {
                    // if our parameter is too long, just overwrite the last character
                    // we should never have this case, but this keeps us from crashing
                    and burning
                    parameter_position = (PARAMETER_MAX_LENGTH - 1);
                }

                // always make the last character a null
                parameters[parameter_index][parameter_position] = CHAR_NULL;
                break;
        }

        buffer_to_parameterize->read_position++;
    }

    return;
}

bool process_data_parameters(char
parameters[PARAMETER_MAX_COUNT][PARAMETER_MAX_LENGTH], struct buffer * send_buffer)
{
    bool end_of_transmission_received = false;


    if (strmatch(parameters[0], "END") == true)
    {
        //  if( LEDSET == 1 )
        //  {
        //      LEDSET = 0;
        //  }
        //  else
        //  {
        //      LEDSET = 1;
```

```
            //  }

            end_of_transmission_received = true;
        }
        else if (strmatch(parameters[0], "Set") == true)
        {
            if (strmatch(parameters[1], "Calibration") == true)
            {
                // set the calibration value for the current sense, if required
            }
            else if (strmatch(parameters[1], "EnUsed") == true)
            {
                // set the Energy used
                // this likely means that the command board had a stored power used greater than we have here.
                // this happens when the power is lost - current sense starts at 0, command board stores in EEPROM

                meterEnergyUsed = atol(parameters[2]);
                com_command_setEnergyUsed(send_buffer);
            }


            //meterEnergyUsed

        }
        else if (strmatch(parameters[0], "Read") == true)
        {
            // nothing to read right now
        }
        else if (strmatch(parameters[0], "Data") == true)
        {
            if (strmatch(parameters[1], "LEDB") == true)
            {
                if (strmatch(parameters[2], "On") == true)
                {
                    command_builder3(send_buffer, "Set", "LEDB", "Off");
                }
                else if (strmatch(parameters[2], "Off") == true)
                {
                    command_builder3(send_buffer, "Set", "LEDB", "On");
                }
            }
        }
        else if (strmatch(parameters[0], "Conf") == true)
        {
            if (strmatch(parameters[1], "LEDB") == true)
            {
                send_end_of_transmission(send_buffer);
            }
            else if (strmatch(parameters[1], "Watts") == true)
            {
                send_end_of_transmission(send_buffer);
            }
            else if (strmatch(parameters[1], "EnUsed") == true)
            {
                send_end_of_transmission(send_buffer);
            }
            else if (strmatch(parameters[1], "Volts") == true)
            {
                send_end_of_transmission(send_buffer);
            }
            else if (strmatch(parameters[1], "Amps") == true)
            {
                send_end_of_transmission(send_buffer);
            }
            else if (strmatch(parameters[1], "PSVersion") == true)
            {
                send_end_of_transmission(send_buffer);
            }
        }

        // add new parameters as necessary
        // NEVER check for a higher parameter number than we allocated for.
        // see earlier comments about NULLS and dying from old age

        return end_of_transmission_received;
    }

    void command_builder1(struct buffer *send_buffer, char* data1)
    {
        command_builder_add_char(send_buffer, COMMAND_SEND_RECEIVE_PRIMER_CHAR);
        command_builder_add_char(send_buffer, COMMAND_SEND_RECEIVE_PRIMER_CHAR);
        command_builder_add_char(send_buffer, COMMAND_START_CHAR);
        command_builder_add_string(send_buffer, data1);
        command_builder_add_char(send_buffer, COMMAND_END_CHAR);

        return;
    }

    void command_builder2(struct buffer *send_buffer, char* data1, char* data2)
    {
        command_builder_add_char(send_buffer, COMMAND_SEND_RECEIVE_PRIMER_CHAR);
        command_builder_add_char(send_buffer, COMMAND_SEND_RECEIVE_PRIMER_CHAR);
        command_builder_add_char(send_buffer, COMMAND_START_CHAR);
        command_builder_add_string(send_buffer, data1);
        command_builder_add_char(send_buffer, COMMAND_DELIMETER);
        command_builder_add_string(send_buffer, data2);
        command_builder_add_char(send_buffer, COMMAND_END_CHAR);

        return;
    }

    void command_builder3(struct buffer *send_buffer, char* data1, char* data2, char* data3)
    {
        command_builder_add_char(send_buffer, COMMAND_SEND_RECEIVE_PRIMER_CHAR);
        command_builder_add_char(send_buffer, COMMAND_SEND_RECEIVE_PRIMER_CHAR);
        command_builder_add_char(send_buffer, COMMAND_START_CHAR);
        command_builder_add_string(send_buffer, data1);
        command_builder_add_char(send_buffer, COMMAND_DELIMETER);
        command_builder_add_string(send_buffer, data2);
        command_builder_add_char(send_buffer, COMMAND_DELIMETER);
        command_builder_add_string(send_buffer, data3);
        command_builder_add_char(send_buffer, COMMAND_END_CHAR);

        return;
    }

    void command_builder4(struct buffer *send_buffer, char* data1, char* data2, char*
    data3, char* data4)
    {
        command_builder_add_char(send_buffer, COMMAND_SEND_RECEIVE_PRIMER_CHAR);
        command_builder_add_char(send_buffer, COMMAND_SEND_RECEIVE_PRIMER_CHAR);
        command_builder_add_char(send_buffer, COMMAND_START_CHAR);
        command_builder_add_string(send_buffer, data1);
        command_builder_add_char(send_buffer, COMMAND_DELIMETER);
        command_builder_add_string(send_buffer, data2);
        command_builder_add_char(send_buffer, COMMAND_DELIMETER);
        command_builder_add_string(send_buffer, data3);
        command_builder_add_char(send_buffer, COMMAND_DELIMETER);
        command_builder_add_string(send_buffer, data4);
        command_builder_add_char(send_buffer, COMMAND_END_CHAR);

        return;
    }

    void command_builder_add_char(struct buffer *send_buffer, char data)
    {
```

```c
    send_buffer->data_buffer[send_buffer->write_position] = data;

    send_buffer->write_position++;
    if (send_buffer->write_position >= BUFFER_LENGTH)
    {
        send_buffer->write_position = 0;
    }

    return;
}

void command_builder_add_string(struct buffer *send_buffer, char *data_string)
{
    for (int inx = 0; data_string[inx] != CHAR_NULL; inx++)
    {
        command_builder_add_char(send_buffer, data_string[inx]);
    }

    return;
}

bool send_data(struct buffer * send_buffer)
{
    bool send_end;

    if (send_buffer->read_position == send_buffer->write_position)
    {
        send_end = true;
        SPI_send_data('\0');
    }
    else
    {
        send_end = false;


        if (SPI_send_data(send_buffer->data_buffer[send_buffer->read_position]) == true)
        {

            send_buffer->read_position++;
            if (send_buffer->read_position >= BUFFER_LENGTH)
            {
                send_buffer->read_position = 0;
            }
        }
    }

    return send_end;
}

bool strmatch(char* a, char* b)
{
    int result;
    bool match;

    result = strcmp2(a, b);

    match = (result == 0) ? true : false;

    return match;
}

int strcmp2(char* a, char* b)
{
    int inx = 0;
    int match = 0;

    while ((a[inx] != CHAR_NULL) && (b[inx] != CHAR_NULL) && (match == 0))
    {
        if (a[inx] > b[inx])
        {
            match = 1;
        }
        else if (a[inx] < b[inx])
        {
            match = -1;
        }
        else if (a[inx] == b[inx])
        {
            //do nothing = never reset to zero
        }

        inx++;
    }


    if ((a[inx] == CHAR_NULL) && (b[inx] != CHAR_NULL))
    {
        match = -1;
    }
    else if ((a[inx] != CHAR_NULL) && (b[inx] == CHAR_NULL))
    {
        match = 1;
    }

    return match;

}

bool SPI_receive_data(char *data)
{

    bool recvGood = false;

    if (SSP2STATbits.BF == 1)
    {
        *data = SSP2BUF;
        recvGood = true;
        SSP2CON1bits.WCOL = 0;
        SPI_transmit_wait = false;
    }
    else
    {
        recvGood = false;
    }

    return recvGood;

}

bool SPI_send_data(char data)
{
    bool sendGood = false;

    if (SPI_transmit_wait == false)
    {
        SSP2BUF = data;
        SPI_transmit_wait = true;
        sendGood = true;
    }
    else
    {
        sendGood = false;
    }

    return sendGood;
}

/************************/
```

```
// RESPONSES

void send_end_of_transmission(struct buffer * send_buffer)
{
    command_builder1(send_buffer, "END");

    return;
}

void com_command_testLED(struct buffer * send_buffer)
{
    command_builder2(send_buffer, "Read", "LEDB");

    return;
}

void com_command_setPower(struct buffer * send_buffer)
{

    char temp[12];

    ultoa(temp, meterWatts, 10);
    command_builder3(send_buffer, "Set", "Watts", temp);

    return;
}

void com_command_setEnergyUsed(struct buffer * send_buffer)
{
    char temp[12];

    ultoa(temp, meterEnergyUsed, 10);

    command_builder3(send_buffer, "Set", "EnUsed", temp);

    return;


}

void com_command_setVolts(struct buffer * send_buffer)
{
    command_builder3(send_buffer, "Set", "Volts", "222");

    return;
}

void com_command_setAmps(struct buffer * send_buffer)
{
    command_builder3(send_buffer, "Set", "Amps", "333");

    return;
}

void com_command_readCalibration(struct buffer * send_buffer)
{
    command_builder2(send_buffer, "Read", "Calibration");

    return;
}

void com_command_setVersion(struct buffer * send_buffer)
{
    command_builder3(send_buffer, "Set", "PSVersion", "444");

}

void SPISlaveInit(void)
{

    TRISAbits.TRISA0 = 0; // pin 2 connected as an output for pulse
    TRISAbits.TRISA1 = 1; // pin 3 connected as an input for pulse
    //    LEDDIR = 0; // pin 25 connected as an output for LED
    TRISCbits.TRISC3 = 0; // pin 14 connected as an output for pulse freq.
    TRISCbits.TRISC5 = 0; // pin 16 connected as an output for pulse freq.
    TRISCbits.TRISC6 = 0; // set pin 17 as an output for MCLR
    TRISCbits.TRISC7 = 0; // set pin 18 as an output for pulse freq.
    ANSELAbits.ANSA1 = 0b0; // turn off analog to digital conversion

    LATCbits.LATC6 = 1; // set the MCLR of the MCP high
    LATCbits.LATC3 = 1; // set pin 14 to a 1 to set freq. control F2 for pulse
    LATCbits.LATC5 = 1; // set pin 16 to a 1 to set freq. control F1 for pulse
    LATCbits.LATC7 = 1; // set pin 18 to a 1 to set freq. control F0 for pulse


    SSP2CON1bits.SSPEN = 0; //Synchronous Serial Port Enable bit

    ANSELBbits.ANSB0 = 0b0;
    ANSELBbits.ANSB1 = 0b0;
    ANSELBbits.ANSB2 = 0b0;
    ANSELBbits.ANSB3 = 0b0;

    TRISBbits.RB0 = 0b1;
    TRISBbits.RB1 = 0b1;
    TRISBbits.RB2 = 0b1;
    TRISBbits.RB3 = 0b0;

    SSP2STATbits.SMP = 0;
    SSP2STATbits.CKE = 1;

    SSP2CON1bits.WCOL = 0; //Write Collision Detect bit
    SSP2CON1bits.SSPOV = 0; //Receive Overflow Indicator bit
    SSP2CON1bits.SSPEN = 0; //Synchronous Serial Port Enable bit
    SSP2CON1bits.CKP = 1; //Clock Polarity Select bit
    SSP2CON1bits.SSPM = 0b0100; //Synchronous Serial Port Mode Select bits


    SSP2CON3 = 0x00;
    SSP2CON3bits.BOEN = 0b0; //Buffer Overwrite Enable bit

    SSP2CON1bits.SSPEN = 1; //Synchronous Serial Port Enable bit

    //    SPIWatchdogTimerInit();

    return;
}

```