Appendix C: C++ source: Arduino TransferProtocol library

```
########################
# keywords.txt
#
# Keyword Definition
########################

ControllerTransferProtocol    KEYWORD1
getSystemVoltage       KEYWORD2
getBatteryVoltage      KEYWORD2
getPwrSrcVoltage       KEYWORD2
getMinBatteryVoltage   KEYWORD2
getChargeCurrent       KEYWORD2
getLoadCurrent KEYWORD2
getBatteryTemp KEYWORD2
getTotalKilowattHrs    KEYWORD2
getTotalAmpHrs KEYWORD2
getAllData     KEYWORD2


/*
  ControllerTransferProtocol.h

  Defines the interface for data transfer over an arbitrary transfer protocol

  NOTE: The methods described here are intended to be fairly generic. A controller
  managing the production of power from any given power source, solar, wind, etc, should
  be able to produce a response to the methods described here. Most any statistic
  about the state of the system can be deduced from this list of critical diagnostic information.
*/

#ifndef ControllerTransferProtocol_h
#define ControllerTransferProtocol_h

#include "WProgram.h"

class ControllerTransferProtocol
{
       public:

              /* Constructor */
              ControllerTransferProtocol();

              /* Get the voltage of the system, e.g. 6V, 12V, 24V, 48V */
              double getSystemVoltage();

              /* Returns the current value of the battery bank voltage */
              double getBatteryVoltage();

              /* Returns the current value of the voltage input to the battery bank */
              double getPwrSrcVoltage();

              /* Returns the cutoff voltage at which loads on the battery bank shut off */
              double getMinBatteryVoltage();

              /* Returns the current value of the current intput to the battery bank */
              double getChargeCurrent();

              /* Returns the current value of the current output of the battery bank */
              double getLoadCurrent();

              /* Returns the current temperature of the battery bank */
              double getBatteryTemp();

              /* Returns the total number of KWH produced thus far */
              double getTotalKilowattHrs();

              /* Returns the total number of AH produced thus far */
```

```
                double getTotalAmpHrs();

                /* Get all data, returns an array of doubles in the order presented:
                        INDEX   VALUE
                        0               System Voltage
                        1               Battery Voltage
                        2               Power Source Voltage
                        3               Minimum Battery Voltage (Shut-Off Point)
                        4               Charge Current
                        5               Load Current
                        6               Battery Temperature
                        7               Total Kilowatt Hours Produced
                        8               Total Amp Hours Produced
                */
                double* getAllData();
};

#endif


/*
  TriStarModbusRTU.cpp
  Implementation of MODBUS RTU protocol for a TS-45 Solar controller
*/

#include "WProgram.h"
#include "ControllerTransferProtocol.h"

/* Method declaration */
void constructAndSendQuery( unsigned char* arr, unsigned short cmd_length );
void sendQuery( unsigned char* arr, unsigned short cmd_length );
double getIncomingValue( double scalar, unsigned short num_bytes_exp );
int readSerialBuffer( unsigned char* arr, unsigned short num_bytes_exp );
bool validateReceivedBuffer( unsigned char* buffer, unsigned short num_bytes );
unsigned short generateCRC_16( unsigned char* data_frame, unsigned short data_length );

/* Constants used in most commands */
static unsigned char TriStarDevAddr = 0x01;
static unsigned char cmdReadHoldingRegs = 0x03;
static unsigned char singleRegHigh = 0x00;
static unsigned char singleRegLow = 0x01;

/* Commands for single values are usually 6 fields long */
static unsigned short stdQueryLen = 6;

/* Commands for single values usually have a 7 field response */
static unsigned short stdResponseLen = 7;

/* TS Modbus registers have scalars used to calculate their equivalent double value */
static double stdScalar1 = 0.002950042724609375;
static double stdScalar2 = 0.00424652099609375;
static double stdScalar3 = 0.002034515380859375;
static double stdScalar4 = 0.00966400146484375;
static double stdScalar5 = 0.000031;
static double stdScalar6 = 0.1;

/*=============================================================================
   COMMAND DEFINITIONS
   ============================================================================
*/

/* Get System Voltage */
static unsigned char systemVoltage[] = {
        TriStarDevAddr,                         // Tri-Star device address
        cmdReadHoldingRegs,             // Function: Read Holding Registers
        0xF0,                                   // Starting Register High
        0x05,                                   // Starting Register Low
        singleRegHigh,                  // No. of Registers High
        singleRegLow                    // No. of Registers Low
};
static double systemVoltageScalar = stdScalar5;
```

```
/* Get Battery Voltage */
static unsigned char batteryVoltage[] = {
       TriStarDevAddr,                        // Tri-Star device address
       cmdReadHoldingRegs,    // Function: Read Holding Registers
       0x00,                                  // Starting Register High
       0x08,                                  // Starting Register Low
       singleRegHigh,                // No. of Registers High
       singleRegLow                           // No. of Registers Low
};
static double batteryVoltageScalar = stdScalar1;

/* Get Power Source Voltage */
static unsigned char powerSourceVoltage[] = {
       TriStarDevAddr,                        // Tri-Star device address
       cmdReadHoldingRegs,    // Function: Read Holding Registers
       0x00,                                  // Starting Register High
       0x0A,                                  // Starting Register Low
       singleRegHigh,                // No. of Registers High
       singleRegLow                           // No. of Registers Low
};
static double powerSourceVoltageScalar = stdScalar2;

/* Get Minimum Battery Voltage */
static unsigned char minBatteryVoltage[] = {
       TriStarDevAddr,                        // Tri-Star device address
       cmdReadHoldingRegs,    // Function: Read Holding Registers
       0xE0,                                  // Starting Register High
       0x2B,                                  // Starting Register Low
       singleRegHigh,                // No. of Registers High
       singleRegLow                           // No. of Registers Low
};
static double minBatteryVoltageScalar = stdScalar1;

/* Get Charge Current */
static unsigned char chargeCurrent[] = {
       TriStarDevAddr,                        // Tri-Star device address
       cmdReadHoldingRegs,    // Function: Read Holding Registers
       0x00,                                  // Starting Register High
       0x0B,                                  // Starting Register Low
       singleRegHigh,                // No. of Registers High
       singleRegLow                           // No. of Registers Low
};
static double chargeCurrentScalar = stdScalar3;

/* Get Load Current */
static unsigned char loadCurrent[] = {
       TriStarDevAddr,                        // Tri-Star device address
       cmdReadHoldingRegs,    // Function: Read Holding Registers
       0x00,                                  // Starting Register High
       0x0C,                                  // Starting Register Low
       singleRegHigh,                // No. of Registers High
       singleRegLow                           // No. of Registers Low
};
static double loadCurrentScalar = stdScalar4;

/* Get Battery Temperature */
static unsigned char batteryTemp[] = {
       TriStarDevAddr,                        // Tri-Star device address
       cmdReadHoldingRegs,    // Function: Read Holding Registers
       0x00,                                  // Starting Register High
       0x0F,                                  // Starting Register Low
       singleRegHigh,                // No. of Registers High
       singleRegLow                           // No. of Registers Low
};
static double batteryTempScalar = 1;

/* Get Total Kilowatt Hours */
static unsigned char totalKilowattHours[] = {
       TriStarDevAddr,                        // Tri-Star device address
       cmdReadHoldingRegs,    // Function: Read Holding Registers
```

```
        0xE0,                                   // Starting Register High
        0x2A,                                   // Starting Register Low
        singleRegHigh,                  // No. of Registers High
        singleRegLow                            // No. of Registers Low
};
static double totalKilowattHoursScalar = 1;

/* Get Total Amp Hours */
static unsigned char totalAmpHours[] = {
        TriStarDevAddr,                         // Tri-Star device address
        cmdReadHoldingRegs,    // Function: Read Holding Registers
        0xE0,                                   // Starting Register High
        0x28,                                   // Starting Register Low
        00,                                             // No. of Registers High
        02                                      // No. of Registers Low
};
static double totalAmpHoursScalar = stdScalar6;

/* CRC High Byte Vector */
static const unsigned char auchCRCHi[] = {
        0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81,
        0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0,
        0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01,
        0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41,
        0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81,
        0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0,
        0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01,
        0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40,
        0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81,
        0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0,
        0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01,
        0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,
        0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81,
        0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0,
        0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01,
        0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,
        0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81,
        0x40
};

/* CRC Low Byte Vector */
static const unsigned char auchCRCLo[] = {
        0x00, 0xC0, 0xC1, 0x01, 0xC3, 0x03, 0x02, 0xC2, 0xC6, 0x06, 0x07, 0xC7, 0x05, 0xC5, 0xC4,
        0x04, 0xCC, 0x0C, 0x0D, 0xCD, 0x0F, 0xCF, 0xCE, 0x0E, 0x0A, 0xCA, 0xCB, 0x0B, 0xC9, 0x09,
        0x08, 0xC8, 0xD8, 0x18, 0x19, 0xD9, 0x1B, 0xDB, 0xDA, 0x1A, 0x1E, 0xDE, 0xDF, 0x1F, 0xDD,
        0x1D, 0x1C, 0xDC, 0x14, 0xD4, 0xD5, 0x15, 0xD7, 0x17, 0x16, 0xD6, 0xD2, 0x12, 0x13, 0xD3,
        0x11, 0xD1, 0xD0, 0x10, 0xF0, 0x30, 0x31, 0xF1, 0x33, 0xF3, 0xF2, 0x32, 0x36, 0xF6, 0xF7,
        0x37, 0xF5, 0x35, 0x34, 0xF4, 0x3C, 0xFC, 0xFD, 0x3D, 0xFF, 0x3F, 0x3E, 0xFE, 0xFA, 0x3A,
        0x3B, 0xFB, 0x39, 0xF9, 0xF8, 0x38, 0x28, 0xE8, 0xE9, 0x29, 0xEB, 0x2B, 0x2A, 0xEA, 0xEE,
        0x2E, 0x2F, 0xEF, 0x2D, 0xED, 0xEC, 0x2C, 0xE4, 0x24, 0x25, 0xE5, 0x27, 0xE7, 0xE6, 0x26,
        0x22, 0xE2, 0xE3, 0x23, 0xE1, 0x21, 0x20, 0xE0, 0xA0, 0x60, 0x61, 0xA1, 0x63, 0xA3, 0xA2,
        0x62, 0x66, 0xA6, 0xA7, 0x67, 0xA5, 0x65, 0x64, 0xA4, 0x6C, 0xAC, 0xAD, 0x6D, 0xAF, 0x6F,
        0x6E, 0xAE, 0xAA, 0x6A, 0x6B, 0xAB, 0x69, 0xA9, 0xA8, 0x68, 0x78, 0xB8, 0xB9, 0x79, 0xBB,
        0x7B, 0x7A, 0xBA, 0xBE, 0x7E, 0x7F, 0xBF, 0x7D, 0xBD, 0xBC, 0x7C, 0xB4, 0x74, 0x75, 0xB5,
        0x77, 0xB7, 0xB6, 0x76, 0x72, 0xB2, 0xB3, 0x73, 0xB1, 0x71, 0x70, 0xB0, 0x50, 0x90, 0x91,
        0x51, 0x93, 0x53, 0x52, 0x92, 0x96, 0x56, 0x57, 0x97, 0x55, 0x95, 0x94, 0x54, 0x9C, 0x5C,
        0x5D, 0x9D, 0x5F, 0x9F, 0x9E, 0x5E, 0x5A, 0x9A, 0x9B, 0x5B, 0x99, 0x59, 0x58, 0x98, 0x88,
        0x48, 0x49, 0x89, 0x4B, 0x8B, 0x8A, 0x4A, 0x4E, 0x8E, 0x8F, 0x4F, 0x8D, 0x4D, 0x4C, 0x8C,
        0x44, 0x84, 0x85, 0x45, 0x87, 0x47, 0x46, 0x86, 0x82, 0x42, 0x43, 0x83, 0x41, 0x81, 0x80,
        0x40
};

/*============================================================================
   PRIVATE CLASS METHODS
   ============================================================================
*/
/* Builds and sends a Modbus query */
void constructAndSendQuery( unsigned char* arr, unsigned short cmd_length )
{
        int i;
        unsigned short CRC;
```

```
        unsigned char messageArray[cmd_length+2];

        /* Get CRC value */
        CRC = generateCRC_16( arr, cmd_length );

        /* Populate message array */
        for( i = 0; i < cmd_length; ++i )
        {
                messageArray[i] = arr[i];
        }

        /* Concatenate CRC onto message Array: This code was deduced from observations of the
            TS-45 RS-232 behavior.  This is different from the canonical implementation. */
        messageArray[i++] = (unsigned char) ( CRC >> 8 );    // grab high byte
        messageArray[i++] = (unsigned char) ( CRC & 0xFF );  // get low byte

        sendQuery( messageArray, (cmd_length+2) );
}

void sendQuery( unsigned char* arr, unsigned short cmd_length )
{
        int i;

        /* Output data frame */
        for( i = 0; i < cmd_length; ++i )
        {
                Serial.print( arr[i] );
        }
}

/* Read an incoming transmission, validate, and retrieve data value */
double getIncomingValue( double scalar, unsigned short num_bytes_exp )
{
        int i = 0;
        unsigned char incoming[num_bytes_exp];
        unsigned char DATA_HI, DATA_LO;
        unsigned short byteCount, dataVal;
        double retVal;

        // get bytes from serial stream
        if(readSerialBuffer( incoming, num_bytes_exp ) < 0)
        {
                return -1; // timeout
        }

        // validate and return the values in the stream
        if(validateReceivedBuffer( incoming, num_bytes_exp ))
        {
                // In Modbus RTU, byte 2 of the message is the number of bytes of data
                byteCount = incoming[2];
                if( byteCount == 0x02 )
                {
                        // only one word of data
                        DATA_HI = incoming[3];
                        DATA_LO = incoming[4];
                        dataVal = (unsigned short) ( DATA_HI << 8 | DATA_LO );
                        return (double) dataVal * scalar;
                }
                else // TODO -- NOT IMPLEMENTED!!!!
                {
                        return -1;
                }
        }
        return -1;
}

/* Reads an expected number of bytes from the serial input stream, returns them in a buffer */
int readSerialBuffer( unsigned char* arr, unsigned short num_bytes_exp )
{
        int i = 0;
        int maxTimeout = 4 + (3 * num_bytes_exp);
```

```
        /* Message timeout: 4 byte lengths + num_bytes_exp byte lengths x 3 */
        while( Serial.available() < num_bytes_exp )
        {
                if( i++ >= maxTimeout ) {
                        Serial.flush();
                        return -1; // error -- TIMEOUT
                }
                delay( 1 ); // wait a ms (roughly the transfer time for 1 byte @ 9600 bps)
                //delay( 10 ); // paranoia
        }

        i = 0;
        while((Serial.available() > 0) && (i < num_bytes_exp))
        {
                arr[i++] = Serial.read(); // read response
        }

        // flush tbe buffer
        Serial.flush();
        return 0;
}

/* validate a received array of chars via a CRC check */
bool validateReceivedBuffer( unsigned char* buffer, unsigned short num_bytes )
{
        unsigned short CRC, testCRC;
        unsigned char CRC_LOW, CRC_HI;
        int i = num_bytes-1;

        /* Check CRC to ensure data integrity */
        CRC_LOW = buffer[ i-- ];
        CRC_HI = buffer[ i ];
        testCRC = ( CRC_HI << 8 | CRC_LOW );
        CRC = generateCRC_16(buffer, (num_bytes-2));

        if( testCRC == CRC )
        {
                return true; // transfer is good!
        }
        return false;
}

/* Generate 16-bit Cyclic Redundancy Check */
unsigned short generateCRC_16( unsigned char* data_frame, unsigned short data_length )
{
        unsigned char  uchCRCHi = 0xFF;              /* start with register being 0xFFFF */
        unsigned char  uchCRCLo = 0xFF;
        unsigned int   index;                        /* index into lookup table */

        while( data_length-- )                       /* calculate CRC */
        {
                /* index gets high byte XOR'd with next item in dataframe   */
                index = uchCRCHi ^ *( data_frame++ );

                /* high byte gets XOR of low byte with high table indexed value */
                uchCRCHi = uchCRCLo ^ auchCRCHi[ index ];

                /* low byte gets low table indexed value */
                uchCRCLo = auchCRCLo[ index ];
        }

        return ( uchCRCHi << 8 | uchCRCLo );  /* combine high and low bytes into CRC, return */
}

/*=============================================================================
   TYPE IMPLEMENTATION
   ============================================================================
*/
/* Constructor */
ControllerTransferProtocol::ControllerTransferProtocol()
```

```
{
	// initialize serial baud rate
	Serial.begin(9600);
}

/* Get the voltage of the system, e.g. 6V, 12V, 24V, 48V */
double ControllerTransferProtocol::getSystemVoltage()
{
	constructAndSendQuery( systemVoltage, stdQueryLen );
	return getIncomingValue( systemVoltageScalar, stdResponseLen );
}

/* Returns the current value of the battery bank voltage */
double ControllerTransferProtocol::getBatteryVoltage()
{
	constructAndSendQuery( batteryVoltage, stdQueryLen );
	return getIncomingValue( batteryVoltageScalar, stdResponseLen );
}

/* Returns the current value of the voltage input to the battery bank */
double ControllerTransferProtocol::getPwrSrcVoltage()
{
	constructAndSendQuery( powerSourceVoltage, stdQueryLen );
	return getIncomingValue( powerSourceVoltageScalar, stdResponseLen );
}

/* Returns the cutoff voltage at which loads on the battery bank should be shut off */
double ControllerTransferProtocol::getMinBatteryVoltage()
{
	constructAndSendQuery( minBatteryVoltage, stdQueryLen );
	return getIncomingValue( minBatteryVoltageScalar, stdResponseLen );
}

/* Returns the current value of the current intput to the battery bank */
double ControllerTransferProtocol::getChargeCurrent()
{
	constructAndSendQuery( chargeCurrent, stdQueryLen );
	return getIncomingValue( chargeCurrentScalar, stdResponseLen );
}

/* Returns the current value of the current output of the battery bank */
double ControllerTransferProtocol::getLoadCurrent()
{
	constructAndSendQuery( loadCurrent, stdQueryLen );
	return getIncomingValue( loadCurrentScalar, stdResponseLen );
}

/* Returns the current temperature of the battery bank */
double ControllerTransferProtocol::getBatteryTemp()
{
	constructAndSendQuery( batteryTemp, stdQueryLen );
	return getIncomingValue( batteryTempScalar, stdResponseLen );
}

/* Returns the total number of KWH produced thus far */
double ControllerTransferProtocol::getTotalKilowattHrs()
{
	constructAndSendQuery( totalKilowattHours, stdQueryLen );
	return getIncomingValue( totalKilowattHoursScalar, stdResponseLen );
}

/* Returns the total number of AH produced thus far */
double ControllerTransferProtocol::getTotalAmpHrs()
{
	constructAndSendQuery( totalAmpHours, stdQueryLen );
	return 0;
}

/* Get all data, returns an array of doubles in the order presented:
	INDEX	VALUE
	0		System Voltage
```

```
        1               Battery Voltage
        2               Power Source Voltage
        3               Minimum Battery Voltage (Shut-Off Point)
        4               Charge Current
        5               Load Current
        6               Battery Temperature
        7               Total Kilowatt Hours Produced
        8               Total Amp Hours Produced
*/
double* ControllerTransferProtocol::getAllData()
{
        double arr[9];
        arr[0] = getSystemVoltage();
        arr[1] = getBatteryVoltage();
        arr[2] = getPwrSrcVoltage();
        arr[3] = getMinBatteryVoltage();
        arr[4] = getChargeCurrent();
        arr[5] = getLoadCurrent();
        arr[6] = getBatteryTemp();
        arr[7] = getTotalKilowattHrs();
        arr[8] = getTotalAmpHrs();
        return arr;
}
```